VISIT…

Thinking in C#
(Revision 0.1)

Bruce Eckel, President,
MindView, Inc.

Planet PDF brings you the Portable Document Format (PDF) version of Thinking in C# (Revision 0.1). Planet PDF is the premier PDF-related site on the web. There is news, software, white papers, interviews, product reviews, Web links, code samples, a forum, and regular articles by many of the most prominent and respected PDF experts in the world. Visit our sites for more detail:

http://www.planetpdf.com/
http://www.pdfstore.com/
http://www.binarything.com/

Thinking in C#

Larry O'Brien
and
Bruce Eckel

ISBN 0-13-027363-5

Thinking in C#

Larry O'Brien
and
Bruce Eckel

Prentice Hall
Upper Saddle River, New Jersey 07458
www.phptr.com

Overview

Introduction 3
1: Those Who Can, Code 15
2: Introduction to Objects 15
3: Hello, Objects 51
4: Controlling Program Flow 89
5: Initialization & Cleanup 151
6: Coupling and Cohesion 215
6a: Hiding the Implementation 234
7: Reusing classes 250
8: Interfaces and Implementation 295
8a: Interfaces 333
9: Collecting Your Objects 391
10: Error Handling With Exceptions 483
11: I/O in C# 521
12: Reflection and Attributes 559
13: Programming Windows Forms 589
14: GDI+ Overview 707
14: Multithreaded Programming 713
15: XML 751
16: Web Services 753
A: C# For Java Programmers 755
B: C# For Visual Basic Programmers 757

C: C# Programming Guidelines 759
D: Resources 771
Index 771

What's Inside

Introduction 3
Prerequisites 3
Learning C# 3
Goals 4
Online documentation 6
Chapters 6
Exercises 9
Source code 10
Coding standards 12
C# versions 12
Seminars and mentoring 12
Errors 12
Note on the cover design 13
Acknowledgements 13
Internet contributors 13
1: Those Who Can, Code 15
2: Introduction to Objects 15
The progress of abstraction 16
An object has an interface 19
An object provides services 22
The hidden implementation 22
Reusing the implementation 24
Inheritance: reusing the interface 25
Is-a vs. is-like-a relationships 29
Interchangeable objects with polymorphism 31
Abstract base classes and interfaces 35
Object landscapes and lifetimes 36
Collections and iterators 37
The singly rooted hierarchy 39
Collection libraries and support for easy collection use 40
The housekeeping dilemma: who should clean up? 41
Exception handling: dealing with errors 43
Multithreading 44
Persistence 45
C# and the Internet 45
What is the Web? 46
Client-side programming 46
Server-side programming 46
A separate arena: applications 46
Analysis and design 47
Extreme programming 47
Why .NET succeeds 47
Systems are easier to express and understand 47
Maximal leverage with libraries 47
Error handling 47
Programming in the large 47

Strategies for transition 48
Guidelines 48
Management obstacles 50
C# vs. Java? 50
Summary 50
3: Hello, Objects 51
You manipulate objects with references 51
You must create all the objects 52
Where storage lives 53
Arrays in Java 54
Special case: value types 55
You never need to destroy an object 56
Scoping 56
Scope of objects 57
Creating new data types: class 58
Fields, Properties, and methods 59
Methods, arguments, and return values 61
The argument list 62
Attributes and Meta-Behavior 64
Delegates 64
Properties 65
Creating New Value Types 67
Enumerations 67
Structs 68
Building a C# program 69
Name visibility 69
Using other components 70
The **static** keyword 71
Putting It All Together 73
Compiling and running 76
Fine-tuning Compilation 77
The Common Language Runtime 77
Comments and embedded documentation 81
Documentation Comments 82
Documentation example 85
Coding style 86
Summary 87
Exercises 87
4: Controlling Program Flow 89
Using C#operators 89
Precedence 90
Assignment 90
C#'s Preprocessor 115
foreach 135
5: Initialization & Cleanup 151
Guaranteed initialization with the constructor 151
Method overloading 154
Distinguishing overloaded methods156
Overloading with primitives 157
Overloading on return values 162
Default constructors 162
The **this** keyword 163
Cleanup: finalization and garbage collection 169
What are destructors for? 171
Instead of a destructor, use Close() or Dispose() 172
Destructors, Dispose(), and the using keyword 177
The death condition 182
How a garbage collector works 183
Member initialization 185
Specifying initialization 187
Constructor initialization 188

Array initialization 195
Multidimensional arrays.....200
Sidebar/Appendix: What a difference a rectangle makes 203
Summary 213
Exercises 214

6: Coupling and Cohesion 215

Software As Architecture vs. Software Architecture 217
What Is Software Architecture..... 219
Simulation Architectures: Always Taught, Rarely Used219
Client/Server and n-Tier Architectures 220
Layered Architectures 222
Problem-Solving Architectures 223
Dispatching Architectures223
"Not Really Object-Oriented"224
Design Is As Design Does224
First, Do No Harm 225
Design Rule #1: Write Boring Code.....226
Design Is As Design Does234

6a: Hiding the Implementation 234

The namespace unit 235
Creating unique package names 236
Using imports to change behavior239
C#'s access specifiers 240
"Friendly" 240
public: interface access 241
private: you can't touch that!... 242
protected: "sort of friendly" 244
Interface and implementation.....245
Class access.....246
Summary 249
Exercises.....250

7: Reusing classes 250

Composition syntax..... 251
Inheritance syntax.....255
Initializing the base class 258
Combining composition and inheritance..... 261
Guaranteeing proper cleanup 263
Choosing composition vs. inheritance.....267
protected.....269
Incremental development270
Upcasting..... 271
Why "upcasting"? 272
Explicit Overloading Only..... 273
The **const** and **readonly** keywords.....286
Sealed classes.....289
Emphasize virtual functions 290
Initialization and class loading 291
Initialization with inheritance.... 291
Summary 293
Exercises.....294

8: Interfaces and Implementation 295

Upcasting revisited.....297
Forgetting the object type 299
The twist 301
Method-call binding.....301
Producing the right behavior 303
Extensibility..... 306
Overriding vs. overloading310

Operator Overloading 311
Abstract classes and methods 311
Constructors and polymorphism 317
Order of constructor calls 317
Behavior of polymorphic methods inside constructors.....320
Designing with inheritance322
Pure inheritance vs. extension... 324
Downcasting and run-time type identification..... 326
Summary 331
Exercises..... 331
8a: Interfaces 333
Interfaces.....333
"Multiple inheritance" in Java... 338
Extending an interface with inheritance 342
Doesn't work in C#. Must have section on enums and structs earlier343
Initializing fields in interfaces ... 346
Nesting interfaces 347
Inner classes 350
Inner classes and upcasting..... 352
Inner classes in methods and scopes 355
Anonymous inner classes 357
The link to the outer class.....361
static inner classes..... 364
Referring to the outer class object366
Reaching outward from a multiply-nested class 368
Inheriting from inner classes..... 369
Can inner classes be overridden?370
Inner class identifiers 372
Why inner classes?..... 373
Inner classes & control frameworks379
Summary 387
Exercises 388
9: Collecting Your Objects 391
Arrays..... 391
Arrays are first-class objects 393
The **Array** class.....398
Array's Static Methods 399
Array element comparisons 402
What? No bubbles? 404
Unsafe Arrays 406
Get things right..... 409
... Then Get Them Fast 413
Array summary..... 420
Introduction to data structures..... 420
Queues and Stacks 421
ArrayList.....424
BitArray 426
Dictionaries 428
Hashtable.....428
ListDictionary..... 431
SortedList 432
String specialists.....433
One Key, Multiple Values.....433
Customizing Hashcode Providers434
String specialists: StringCollection and StringDictionary.....436
Container disadvantage: unknown type.....437
Using **CollectionBase** to make type-conscious collections..... 440
IEnumerators 442
Custom Indexers 444

Custom Enumerators & Data Structures 448
Sorting and searching **List**s 454
From Collections to Arrays 456
Persistent Data With ADO.NET 463
Getting a handle on data with DataSet 464
Connecting to a database 468
Fast Reading With an IDataReader 472
CRUD With ADO.NET 474
Update and Delete 474
The Object-Relational Impedance Mismatch 479
Summary 480
Exercises 481

10: Error Handling With Exceptions 483
Basic exceptions 487
Exception arguments 488
Catching an exception 489
The **try** block 489
Exception handlers 490
Exceptions have a helplink 491
Creating your own exceptions 491
C#'s Lack Of Checked Exceptions 497
Catching any exception 499
Rethrowing an exception 499
Elevating the abstraction level 500
Standard C# exceptions 502
Performing cleanup with finally 503
What's **finally** for? 505
Finally and **using** 508
Pitfall: the lost exception 510
Constructors 512
Exception matching 516
Exception guidelines 518
Summary 518
@todo – New Chapter? Design By Contract 519
Exercises 519

11: I/O in C# 521
File, **Directory**, and **Path** 521
A directory lister 521
Checking for and creating directories 523
Isolated Stores 525
Input and output 526
Types of **Stream** 527
Text and Binary 528
Working With Different Sources 529
Fun With CryptoStreams 532
BinaryReader and BinaryWriter 536
StreamReader and StreamWriter 541
Random access with Seek 544
Standard I/O 546
Reading from standard input 546
Redirecting standard I/O 547
Regular Expressions 548
Checking capitalization style 553
Summary 557
Exercises 557

12: Reflection and Attributes 559
The need for RTTI 559
The **Class** object 562
Checking before a cast 565
RTTI syntax 574
Reflection: run-time class information 577
A class method extractor 579

Summary 585
Exercises 586

13: Programming Windows Forms 589

Delegates 590
Designing With Delegates592
Multicast Delegates 594
Events 598
Recursive Traps..... 601
The Genesis of Windows Forms.....603
Creating a Form.....605
GUI Architectures 606
Using the Visual Designer606
Form-Event-Control 613
Presentation-Abstraction-Control..... 617
Model-View-Controller ... 621
Layout 626
Non-Code Resources.....630
Creating Satellite Assemblies 636
Constant Resources.....637
What About the XP Look?639
Fancy Buttons..... 641
Tooltips.....645
Displaying & Editing Text646
Linking Text 650
Checkboxes and RadioButtons.....652
List, Combo, and CheckedListBoxes 655
Multiplane displays with the **Splitter** control 661
TreeView & ListView.....663
ListView.....665
Icon Views..... 665
Details View 665
Clipboard and Drag-and-Drop 669
Clipboard 669
Drag and Drop 672
Data-bound Controls 682
Editing Data from Bound Controls 687
Menus 695
Standard Dialogs 699
Usage-Centered Design...702
Summary 703
Exercises 705

14: GDI+ Overview 707

Drawing pixels.....707
Drawing shapes 707
Filling and stroking 707
Printing.....707
Accessing DirectX..... 710
Creating a screensvaer 710
Creating a system service 710
Creating an application (Windows & Menus) 710
Accessible Object**Error! Bookmark not defined.**
Ambient Properties 710
Application 710
ApplicationContext..... 710
AxHost 710
Binding**Error! Bookmark not defined.**
Color Dialog**Error! Bookmark not defined.**
ComboBox**Error! Bookmark not defined.**
CommonDialog**Error! Bookmark not defined.**
ContainerControl**Error! Bookmark not defined.**
Control /ControlEvents**Error! Bookmark not defined.**
ControlPaint**Error! Bookmark not defined.**
CurrencyManager**Error! Bookmark not defined.**
Cursor**Error! Bookmark not defined.**
DataGrid**Error! Bookmark not defined.**

DateTimePicker**Error! Bookmark not defined.**
DomainUpDown**Error! Bookmark not defined.**
Drag and Drop **DoneError! Bookmark not defined.**
ErrorProvider....710
FeatureSupport....710
FileDialog**Error! Bookmark not defined.**
FontDialog**Error! Bookmark not defined.**
Form**Error! Bookmark not defined.**
GDI+**Error! Bookmark not defined.**
GroupBox**Error! Bookmark not defined.**
Help....710
HScrollbar (Scroll bars)**Error! Bookmark not defined.**
ImageList**Error! Bookmark not defined.**
Handling Key Presses**Error! Bookmark not defined.**
Label **DoneError! Bookmark not defined.**
LinkLabels **DoneError! Bookmark no**
ListBox**Error! Bookmark not defined**
ListView**Error! Bookmark not defined.**
Menus **DoneError! Bookmark not defined.**
Message....710
MessageBox....710
MonthCalendar**Error! Bookmark not defined.**
NotifyIcon....710
OpenFileDialog**Error! Bookmark not defined.**
PageSetupDialog**Error! Bookmark not defined.**
Panel**Error! Bookmark not defined.**
PictureBox**Error! Bookmark not defined.**
PrintDialog / Printing**Error! Bookmark not defined.**
Progress Bar....711
PropertyGrid....711
RadioButton**Error! Bookmark not defined.**
RichTextBox **DoneError! Bookmark not defined.**
SaveDialog**Error! Bookmark not defined.**
SelectionRange....711
Splitter**Error! Bookmark not defined.**
StatusBar**Error! Bookmark not defined.**
TabControl/ Tabbed Pages....711
TextBox **DoneError! Bookmark not defined.**
Timer....711
ToolBar....711
ToolTip**Error! Bookmark not defined.**
TrackBar....711
TreeView**Error! Bookmark not defined.**
UserControl....711
Windows Controls**Error! Bookmark not defined.**
Windows Services....711
Programming techniques....711
Binding events dynamically....711
Separating business logic from UI Logic....711
Visual programming....711
Summary....711
Exercises....711
14: Multithreaded Programming 713
.NET's Threading Model....714
Thread Scheduling....714
Threading Problems....714
The Cardinal Rules of Threading....714
Thread Lifecycle....714
Starting Threads....714
Stopping Threads....714
Pausing and Restarting....714
Blocking and Waiting....714
Exception Handling in Threads....714
Threads and Interoperability714
Threads and Garbage Collection....715
Threads and Scalability....715
Responsive user interfaces715
Creating Threads....718
Threading for a responsive interface721
Sharing limited resources723

Improperly accessing resources 723
Using the Monitor class to prevent collisions @todo – confirm mechanism of Monitor and add Mutex sample code and Interlocked730
Threads, Delegates, and Events. 749

15: XML 751
Schemas and DataSets 751

16: Web Services 753

A: C# For Java Programmers 755

B: C# For Visual Basic Programmers 757

C: C# Programming Guidelines 759
Design 759
Implementation.....766

D: Resources 771
Software.....771
Books 771
C#.....771
Analysis & design.....771
Management & Process.....771

Index 771

Introduction

Prerequisites

This book assumes that you have some programming familiarity: you understand that a program is a collection of statements, the idea of a subroutine/function/macro, control statements such as "if" and looping constructs such as "while," etc. However, you might have learned this in many places, such as programming with a macro language or working with a tool like Perl. As long as you've programmed to the point where you feel comfortable with the basic ideas of programming, you'll be able to work through this book. Of course, the book will be *easier* for the C programmers and more so for the C++ programmers, but don't count yourself out if you're not experienced with those languages (but come willing to work hard; also, the multimedia CD that accompanies this book will bring you up to speed on the basic C syntax necessary to learn C#). I'll be introducing the concepts of object-oriented programming (OOP) and C#'sbasic control mechanisms, so you'll be exposed to those, and the first exercises will involve the basic control-flow statements.

Although references will often be made to C and C++ language features, these are not intended to be insider comments, but instead to help all programmers put C# in perspective with those languages, from which, after all, C# is descended. I will attempt to make these references simple and to explain anything that I think a non- C/C++ programmer would not be familiar with.

Learning C#

Tk. At about the same time that my first book *Using C++* (Osborne/McGraw-Hill, 1989) came out, I began teaching that language. Teaching programming languages has become my profession; I've seen nodding heads, blank faces, and puzzled expressions in audiences all over the world since 1989. As I began giving in-house training with smaller groups of people, I discovered something during the exercises. Even those

people who were smiling and nodding were confused about many issues. I found out, by chairing the C++ track at the Software Development Conference for a number of years (and later the Java track), that I and other speakers tended to give the typical audience too many topics too fast. So eventually, through both variety in the audience level and the way that I presented the material, I would end up losing some portion of the audience. Maybe it's asking too much, but because I am one of those people resistant to traditional lecturing (and for most people, I believe, such resistance results from boredom), I wanted to try to keep everyone up to speed.

For a time, I was creating a number of different presentations in fairly short order. Thus, I ended up learning by experiment and iteration (a technique that also works well in C# program design). Eventually I developed a course using everything I had learned from my teaching experience—one that I would be happy giving for a long time. It tackles the learning problem in discrete, easy-to-digest steps, and in a hands-on seminar (the ideal learning situation) there are exercises following each of the short lessons. I now give this course in public C# seminars, which you can find out about at *www.BruceEckel.com*. (The introductory seminar is also available as a CD ROM. Information is available at the same Web site.)

The feedback that I get from each seminar helps me change and refocus the material until I think it works well as a teaching medium. But this book isn't just seminar notes—I tried to pack as much information as I could within these pages, and structured it to draw you through onto the next subject. More than anything, the book is designed to serve the solitary reader who is struggling with a new programming language.

Goals

Tk. Like my previous book *Thinking in C++*, this book has come to be structured around the process of teaching the language. In particular, my motivation is to create something that provides me with a way to teach the language in my own seminars. When I think of a chapter in the book, I think in terms of what makes a good lesson during a seminar. My goal is to get bite-sized pieces that can be taught in a reasonable amount of time,

followed by exercises that are feasible to accomplish in a classroom situation.

My goals in this book are to:

1. Present the material one simple step at a time so that you can easily digest each concept before moving on.

2. Use examples that are as simple and short as possible. This sometimes prevents me from tackling "real world" problems, but I've found that beginners are usually happier when they can understand every detail of an example rather than being impressed by the scope of the problem it solves. Also, there's a severe limit to the amount of code that can be absorbed in a classroom situation. For this I will no doubt receive criticism for using "toy examples," but I'm willing to accept that in favor of producing something pedagogically useful.

3. Carefully sequence the presentation of features so that you aren't seeing something that you haven't been exposed to. Of course, this isn't always possible; in those situations, a brief introductory description is given.

4. Give you what I think is important for you to understand about the language, rather than everything I know. I believe there is an information importance hierarchy, and that there are some facts that 95 percent of programmers will never need to know and that just confuse people and adds to their perception of the complexity of the language. To take an example from C, if you memorize the operator precedence table (I never did), you can write clever code. But if you need to think about it, it will also confuse the reader/maintainer of that code. So forget about precedence, and use parentheses when things aren't clear.

5. Keep each section focused enough so that the lecture time—and the time between exercise periods—is small. Not only does this keep the audience's minds more active and involved during a hands-on seminar, but it gives the reader a greater sense of accomplishment.

6. Provide you with a solid foundation so that you can understand the issues well enough to move on to more difficult coursework and books.

Online documentation

tk

Chapters

This book was designed with one thing in mind: the way people learn the C# language. Seminar audience feedback helped me understand the difficult parts that needed illumination. In the areas where I got ambitious and included too many features all at once, I came to know—through the process of presenting the material—that if you include a lot of new features, you need to explain them all, and this easily compounds the student's confusion. As a result, I've taken a great deal of trouble to introduce the features as few at a time as possible.

The goal, then, is for each chapter to teach a single feature, or a small group of associated features, without relying on additional features. That way you can digest each piece in the context of your current knowledge before moving on.

Here is a brief description of the chapters contained in the book, which correspond to lectures and exercise periods in my hands-on seminars.

Chapter 1: ***Introduction to Objects***
tk

Chapter 2: ***Everything is an Object***
tk

Chapter 3: ***Controlling Program Flow***
tk

Chapter 4: ***Initialization & Cleanup***
This chapter begins by introducing the constructor, which guarantees proper initialization. The definition of the

constructor leads into the concept of function overloading (since you might want several constructors). This is followed by a discussion of the process of cleanup, which is not always as simple as it seems. Normally, you just drop an object when you're done with it and the garbage collector eventually comes along and releases the memory. This portion explores the garbage collector and some of its idiosyncrasies. The chapter concludes with a closer look at how things are initialized: automatic member initialization, specifying member initialization, the order of initialization, **static** initialization and array initialization.

Chapter 5: Hiding the Implementation

tk

Chapter 6: Reusing Classes

The concept of inheritance is standard in virtually all OOP languages. It's a way to take an existing class and add to its functionality (as well as change it, the subject of Chapter 7). Inheritance is often a way to reuse code by leaving the "base class" the same, and just patching things here and there to produce what you want. However, inheritance isn't the only way to make new classes from existing ones. You can also embed an object inside your new class with *composition*. In this chapter you'll learn about these two ways to reuse code in Java, and how to apply them.

Chapter 7: Polymorphism

On your own, you might take nine months to discover and understand polymorphism, a cornerstone of OOP. Through small, simple examples you'll see how to create a family of types with inheritance and manipulate objects in that family through their common base class. C#'s polymorphism allows you to treat all objects in this family generically, which means the bulk of your code doesn't rely on specific type information. This makes your programs extensible, so building programs and code maintenance is easier and cheaper.

Chapter 8: Interfaces

C# provides a third way to set up a reuse relationship, through the *interface*, which is a pure abstraction of the interface of an object. The **interface** is more than just an abstract class taken to the extreme, since it allows you to perform a variation on C++'s "multiple inheritance," by creating a class that can be upcast to more than one base type.

Chapter 9: Holding your Objects

It's a fairly simple program that has only a fixed quantity of objects with known lifetimes. In general, your programs will always be creating new objects at a variety of times that will be known only while the program is running. In addition, you won't know until run-time the quantity or even the exact type of the objects you need. To solve the general programming problem, you need to create any number of objects, anytime, anywhere. This chapter explores in depth the Collection Library that .NET supplies to hold objects while you're working with them: the simple arrays and more sophisticated containers (data structures). This chapter also covers ADO.NET basics.

Chapter 10: Error Handling with Exceptions

The basic philosophy of C# is that badly-formed code will not be run. As much as possible, the compiler catches problems, but sometimes the problems—either programmer error or a natural error condition that occurs as part of the normal execution of the program—can be detected and dealt with only at run-time. C# has *exception handling* to deal with any problems that arise while the program is running. This chapter examines how the keywords **try**, **catch**, **throw**, **throws**, and **finally** work in C#; when you should throw exceptions and what to do when you catch them. In addition, you'll see Java's standard exceptions, how to create your own, what happens with exceptions in constructors, and how exception handlers are located.

Chapter 11: The C# I/O System

Theoretically, you can divide any program into three parts: input, process, and output. This implies that I/O (input/output) is an important part of the equation. In this chapter you'll learn about the different classes that C# provides for reading and writing files, blocks of memory, and the console.

Chapter 12: Run-Time Type Identification

tk

Chapter 13: Programming Windows Applications

tk

Chapter 14: Multiple Threads

C# provides a built-in facility to support multiple concurrent subtasks, called *threads*, running within a single program. (Unless you have multiple processors on your machine, this is only the *appearance* of multiple subtasks.) This chapter looks at the syntax and semantics of multithreading in C#.

Chapter 15: XML

Tk

Chapter 16: Web Services

Appendix A: C# For Java Programmers

tk

Appendix B: C# For Visual Basic Programmers

tk

Exercises

I've discovered that simple exercises are exceptionally useful to complete a student's understanding during a seminar, so you'll find a set at the end of each chapter.

Most exercises are designed to be easy enough that they can be finished in a reasonable amount of time in a classroom situation while the instructor observes, making sure that all the students are absorbing the material.

Some exercises are more advanced to prevent boredom for experienced students. The majority are designed to be solved in a short time and test and polish your knowledge. Some are more challenging, but none present major challenges. (Presumably, you'll find those on your own—or more likely they'll find you).

Source code

All the source code for this book is available as copyrighted freeware, distributed as a single package, by visiting the Web site *www.thinkingin.net*. To make sure that you get the most current version, this is the official site for distribution of the code and the electronic version of the book. You can find mirrored versions of the electronic book and the code on other sites (some of these sites are found at *www.thinkingin.net*), but you should check the official site to ensure that the mirrored version is actually the most recent edition. You may distribute the code in classroom and other educational situations.

The primary goal of the copyright is to ensure that the source of the code is properly cited, and to prevent you from republishing the code in print media without permission. (As long as the source is cited, using examples from the book in most media is generally not a problem.)

In each source code file you will find a reference to the following copyright notice:

```
//:! :CopyRight.txt
Copyright ©2002 Larry O'Brien
Source code file from the 1st edition of the book
"Thinking in C#." All rights reserved EXCEPT as
allowed by the following statements:
You can freely use this file
for your own work (personal or commercial),
including modifications and distribution in
executable form only. Permission is granted to use
this file in classroom situations, including its
use in presentation materials, as long as the book
"Thinking in C#" is cited as the source.
Except in classroom situations, you cannot copy
and distribute this code; instead, the sole
```

```
distribution point is http://www.thinkingin.net
(and official mirror sites) where it is
freely available. You cannot remove this
copyright and notice. You cannot distribute
modified versions of the source code in this
package. You cannot use this file in printed
media without the express permission of the
author. Larry O'Brien makes no representation about
the suitability of this software for any purpose.
It is provided "as is" without express or implied
warranty of any kind, including any implied
warranty of merchantability, fitness for a
particular purpose or non-infringement. The entire
risk as to the quality and performance of the
software is with you. Larry O'Brien, Bruce Eckel, and
the
publisher shall not be liable for any damages
suffered by you or any third party as a result of
using or distributing software. In no event will
Larry O'Brien, Bruce Eckel or the publisher be liable
for any
lost revenue, profit, or data, or for direct,
indirect, special, consequential, incidental, or
punitive damages, however caused and regardless of
the theory of liability, arising out of the use of
or inability to use software, even if Larry O'Brien,
Bruce Eckel
and the publisher have been advised of the
possibility of such damages. Should the software
prove defective, you assume the cost of all
necessary servicing, repair, or correction. If you
think you've found an error, please submit the
correction using the form you will find at
www.thinkingin.net. (Please use the same
form for non-code errors found in the book.)
///:~
```

You may use the code in your projects and in the classroom (including your presentation materials) as long as the copyright notice that appears in each source file is retained.

Coding standards

In the text of this book, identifiers (function, variable, and class names) are set in **bold**. Most keywords are also set in bold, except for those keywords that are used so much that the bolding can become tedious, such as "class."

tk

The programs in this book are files that are included by the word processor in the text, directly from compiled files. Thus, the code files printed in the book should all work without compiler errors. The errors that *should* cause compile-time error messages are commented out with the comment **//!** so they can be easily discovered and tested using automatic means. Errors discovered and reported to the author will appear first in the distributed source code and later in updates of the book (which will also appear on the Web site *www.thinkingin.net*).

C# versions

tk

Seminars and mentoring

tk

Errors

No matter how many tricks a writer uses to detect errors, some always creep in and these often leap off the page for a fresh reader.

There is an error submission form linked from the beginning of each chapter in the HTML version of this book (and on the CD ROM bound into the back of this book, and downloadable from *www.thinkingin.net*) and also on the Web site itself, on the page for this book. If you discover anything you believe to be an error, please use this form to submit the error along with your suggested correction. If necessary, include the

original source file and note any suggested modifications. Your help is appreciated.

Note on the cover design

tk

Acknowledgements

tk

Internet contributors

tk

1: Those Who Can, Code

2: Introduction to Objects

The genesis of the computer revolution was in a machine. The genesis of our programming languages thus tends to look like that machine.

But computers are not so much machines as they are mind amplification tools ("bicycles for the mind," as Steve Jobs is fond of saying) and a different kind of expressive medium. As a result, the tools are beginning to look less like machines and more like parts of our minds, and also like other forms of expression such as writing, painting, sculpture, animation, and filmmaking. Object-oriented programming (OOP) is part of this movement toward using the computer as an expressive medium.

This chapter will introduce you to the basic concepts of OOP, including an overview of development methods. This chapter, and this book, assume that you have had experience in a procedural programming language, although not necessarily C. If you think you need more preparation in programming and the syntax of C before tackling this book, you should work through the *Thinking in C: Foundations for C++ and Java* training CD ROM available at *www.BruceEckel.com*.

This chapter is background and supplementary material. Many people do not feel comfortable wading into object-oriented programming without understanding the big picture first. Thus, there are many concepts that

are introduced here to give you a solid overview of OOP. However, many other people don't get the big picture concepts until they've seen some of the mechanics first; these people may become bogged down and lost without some code to get their hands on. If you're part of this latter group and are eager to get to the specifics of the language, feel free to jump past this chapter—skipping it at this point will not prevent you from writing programs or learning the language. However, you will want to come back here eventually to fill in your knowledge so you can understand why objects are important and how to design with them.

The progress of abstraction

All programming languages provide abstractions. It can be argued that the complexity of the problems you're able to solve is directly related to the kind and quality of abstraction. By "kind" I mean, "What is it that you are abstracting?" Assembly language is a small abstraction of the underlying machine. Many so-called "imperative" languages that followed (such as Fortran, BASIC, and C) were abstractions of assembly language. These languages are big improvements over assembly language, but their primary abstraction still requires you to think in terms of the structure of the computer rather than the structure of the problem you are trying to solve. The programmer must establish the association between the machine model (in the "solution space," which is the place where you're modeling that problem, such as a computer) and the model of the problem that is actually being solved (in the "problem space," which is the place where the problem exists). The effort required to perform this mapping, and the fact that it is extrinsic to the programming language, produces programs that are difficult to write and expensive to maintain, and as a side effect created the entire "programming methods" industry.

The alternative to modeling the machine is to model the problem you're trying to solve. Early languages such as LISP and APL chose particular views of the world ("All problems are ultimately lists" or "All problems are algorithmic," respectively). PROLOG casts all problems into chains of decisions. Languages have been created for constraint-based

programming and for programming exclusively by manipulating graphical symbols. (The latter proved to be too restrictive.) Each of these approaches is a good solution to the particular class of problem they're designed to solve, but when you step outside of that domain they become awkward.

The object-oriented approach goes a step further by providing tools for the programmer to represent elements in the problem space. This representation is general enough that the programmer is not constrained to any particular type of problem. We refer to the elements in the problem space and their representations in the solution space as "objects." (Of course, you will also need other objects that don't have problem-space analogs.) The idea is that the program is allowed to adapt itself to the lingo of the problem by adding new types of objects, so when you read the code describing the solution, you're reading words that also express the problem. This is a more flexible and powerful language abstraction than what we've had before. Thus, OOP allows you to describe the problem in terms of the problem, rather than in terms of the computer where the solution will run. There's still a connection back to the computer, though. Each object looks quite a bit like a little computer; it has a state, and it has operations that you can ask it to perform. However, this doesn't seem like such a bad analogy to objects in the real world—they all have characteristics and behaviors.

Some language designers have decided that object-oriented programming by itself is not adequate to easily solve all programming problems, and advocate the combination of various approaches into *multiparadigm* programming languages.[1]

Alan Kay summarized five basic characteristics of Smalltalk, the first successful object-oriented language and one of the languages upon which C# is based. These characteristics represent a pure approach to object-oriented programming:

1. **Everything is an object.** Think of an object as a fancy variable; it stores data, but you can "make requests" to that object,

[1] See *Multiparadigm Programming in Leda* by Timothy Budd (Addison-Wesley 1995).

asking it to perform operations on itself. In theory, you can take any conceptual component in the problem you're trying to solve (dogs, buildings, services, etc.) and represent it as an object in your program.

2. **A program is a bunch of objects telling each other what to do by sending messages**. To make a request of an object, you "send a message" to that object. More concretely, you can think of a message as a request to call a function that belongs to a particular object.

3. **Each object has its own memory made up of other objects**. Put another way, you create a new kind of object by making a package containing existing objects. Thus, you can build complexity in a program while hiding it behind the simplicity of objects.

4. **Every object has a type**. Using the parlance, each object is an *instance* of a *class*, in which "class" is synonymous with "type." The most important distinguishing characteristic of a class is "What messages can you send to it?"

5. **All objects of a particular type can receive the same messages**. This is actually a loaded statement, as you will see later. Because an object of type "circle" is also an object of type "shape," a circle is guaranteed to accept shape messages. This means you can write code that talks to shapes and automatically handle anything that fits the description of a shape. This *substitutability* is one of the most powerful concepts in OOP.

Booch offers an even more succinct description of an object:

An object has state, behavior and identity

This means that an object can have internal data (which gives it state), methods (to produce behavior), and each object can be uniquely

distinguished from every other object – to put this in a concrete sense, each object has a unique address in memory[2]

An object has an interface

Aristotle was probably the first to begin a careful study of the concept of *type;* he spoke of "the class of fishes and the class of birds." The idea that all objects, while being unique, are also part of a class of objects that have characteristics and behaviors in common was used directly in the first object-oriented language, Simula-67, with its fundamental keyword **class** that introduces a new type into a program.

Simula, as its name implies, was created for developing simulations such as the classic "bank teller problem." In this, you have a bunch of tellers, customers, accounts, transactions, and units of money—a lot of "objects." Objects that are identical except for their state during a program's execution are grouped together into "classes of objects" and that's where the keyword **class** came from. Creating abstract data types (classes) is a fundamental concept in object-oriented programming. Abstract data types work almost exactly like built-in types: You can create variables of a type (called *objects* or *instances* in object-oriented parlance) and manipulate those variables (called *sending messages* or *requests;* you send a message and the object figures out what to do with it). The members (elements) of each class share some commonality: every account has a balance, every teller can accept a deposit, etc. At the same time, each member has its own state, each account has a different balance, each teller has a name. Thus, the tellers, customers, accounts, transactions, etc., can each be represented with a unique entity in the computer program. This entity is the object, and each object belongs to a particular class that defines its characteristics and behaviors.

So, although what we really do in object-oriented programming is create new data types, virtually all object-oriented programming languages use

---

[2] This is actually a bit restrictive, since objects can conceivably exist in different machines and address spaces, and they can also be stored on disk. In these cases, the identity of the object must be determined by something other than memory address.

the “class” keyword. When you see the word “type” think “class” and vice versa[3].

Since a class describes a set of objects that have identical characteristics (data elements) and behaviors (functionality), a class is really a data type because a floating point number, for example, also has a set of characteristics and behaviors. The difference is that a programmer defines a class to fit a problem rather than being forced to use an existing data type that was designed to represent a unit of storage in a machine. You extend the programming language by adding new data types specific to your needs. The programming system welcomes the new classes and gives them all the care and type-checking that it gives to built-in types.

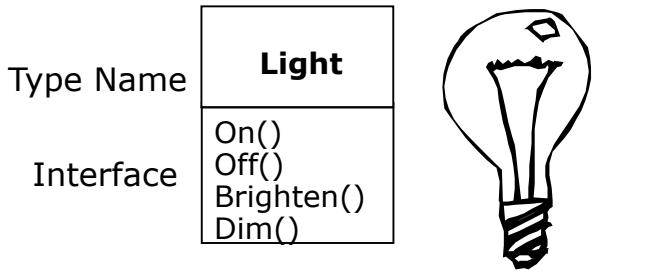
The object-oriented approach is not limited to building simulations. Whether or not you agree that any program is a simulation of the system you’re designing, the use of OOP techniques can easily reduce a large set of problems to a simple solution.

Once a class is established, you can make as many objects of that class as you like, and then manipulate those objects as if they are the elements that exist in the problem you are trying to solve. Indeed, one of the challenges of object-oriented programming is to create a one-to-one mapping between the elements in the problem space and objects in the solution space.

But how do you get an object to do useful work for you? There must be a way to make a request of the object so that it will do something, such as complete a transaction, draw something on the screen, or turn on a switch. And each object can satisfy only certain requests. The requests you can make of an object are defined by its *interface,* and the type is what determines the interface. A simple example might be a representation of a light bulb:

---

[3] Some people make a distinction, stating that type determines the interface while class is a particular implementation of that interface.




```
Light lt = new Light();
lt.On();
```

The interface establishes *what* requests you can make for a particular object. However, there must be code somewhere to satisfy that request. This, along with the hidden data, comprises the *implementation*. From a procedural programming standpoint, it's not that complicated. A type has a function associated with each possible request, and when you make a particular request to an object, that function is called. This process is usually summarized by saying that you "send a message" (make a request) to an object, and the object figures out what to do with that message (it executes code).

Here, the name of the type/class is **Light**, the name of this particular **Light** object is **lt**, and the requests that you can make of a **Light** object are to turn it on, turn it off, make it brighter, or make it dimmer. You create a **Light** object by defining a "reference" (**lt**) for that object and calling **new** to request a new object of that type. To send a message to the object, you state the name of the object and connect it to the message request with a period (dot). From the standpoint of the user of a predefined class, that's pretty much all there is to programming with objects.

The diagram shown above follows the format of the *Unified Modeling Language* (UML). Each class is represented by a box, with the type name in the top portion of the box, any data members that you care to describe in the middle portion of the box, and the *member functions* (the functions that belong to this object, which receive any messages you send to that object) in the bottom portion of the box. Often, only the name of the class and the public member functions are shown in UML design diagrams, and

so the middle portion is not shown. If you're interested only in the class name, then the bottom portion doesn't need to be shown, either.

An object provides services

tk

The hidden implementation

It is helpful to break up the playing field into *class creators* (those who create new data types) and *client programmers* (the class consumers who use the data types in their applications). The goal of the client programmer is to collect a toolbox full of classes to use for rapid application development. The goal of the class creator is to build a class that exposes only what's necessary to the client programmer and keeps everything else hidden. Why? Because if it's hidden, the client programmer can't use it, which means that the class creator can change the hidden portion at will without worrying about the impact on anyone else. The hidden portion usually represents the tender insides of an object that could easily be corrupted by a careless or uninformed client programmer, so hiding the implementation reduces program bugs. The concept of implementation hiding cannot be overemphasized.

In any relationship it's important to have boundaries that are respected by all parties involved. When you create a library, you establish a relationship with the client programmer, who is also a programmer, but one who is putting together an application by using your library, possibly to build a bigger library.

If all the members of a class are available to everyone, then the client programmer can do anything with that class and there's no way to enforce rules. Even though you might really prefer that the client programmer not

directly manipulate some of the members of your class, without access control there's no way to prevent it. Everything's naked to the world.

So the first reason for access control is to keep client programmers' hands off portions they shouldn't touch—parts that are necessary for the internal machinations of the data type but not part of the interface that users need in order to solve their particular problems. This is actually a service to users because they can easily see what's important to them and what they can ignore.

The second reason for access control is to allow the library designer to change the internal workings of the class without worrying about how it will affect the client programmer. For example, you might implement a particular class in a simple fashion to ease development, and then later discover that you need to rewrite it in order to make it run faster. If the interface and implementation are clearly separated and protected, you can accomplish this easily.

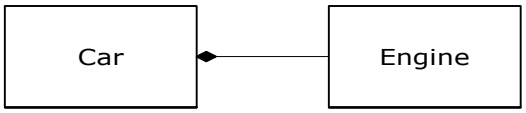
Java uses five explicit keywords to set the boundaries in a class: **public**, **private**, **protected**, **internal,** and **protected internal**. Their use and meaning are quite straightforward. These *access specifiers* determine who can use the definitions that follow. **public** means the following definitions are available to everyone. The **private** keyword, on the other hand, means that no one can access those definitions except you, the creator of the type, inside member functions of that type. **private** is a brick wall between you and the client programmer. If someone tries to access a **private** member, they'll get a compile-time error. **protected** acts like **private**, with the exception that an inheriting class has access to **protected** members, but not **private** members. Inheritance will be introduced shortly. **internal** is often called "friendly"–the definition can be accessed by other classes in the same namespace as if it were **public**, but is not accessible to classes in different namespaces. Namespaces will be discussed in depth in chapter #ref#. **protected internal** allows access by classes within the same namespace (as with **internal**) *or* by inheriting classes (as with **protected**) even if the inheriting classes are not within the same namespace.

C#'s default access, which comes into play if you don't use one of the aforementioned specifiers, is **internal**.

Reusing the implementation

Once a class has been created and tested, it should (ideally) represent a useful unit of code. It turns out that this reusability is not nearly so easy to achieve as many would hope; it takes experience and insight to produce a good design. But once you have such a design, it begs to be reused. Code reuse is one of the greatest advantages that object-oriented programming languages provide.

The simplest way to reuse a class is to just use an object of that class directly, but you can also place an object of that class inside a new class. We call this "creating a member object." Your new class can be made up of any number and type of other objects, in any combination that you need to achieve the functionality desired in your new class. Because you are composing a new class from existing classes, this concept is called *composition* (or more generally, *aggregation*). Composition is often referred to as a "has-a" relationship, as in "a car has an engine."



(The above UML diagram indicates composition with the filled diamond, which states there is one car. I will typically use a simpler form: just a line, without the diamond, to indicate an association.[4])

Composition comes with a great deal of flexibility. The member objects of your new class are usually private, making them inaccessible to the client programmers who are using the class. This allows you to change those members without disturbing existing client code. You can also change the member objects at run-time, to dynamically change the behavior of your program. Inheritance, which is described next, does not have this

[4] This is usually enough detail for most diagrams, and you don't need to get specific about whether you're using aggregation or composition.

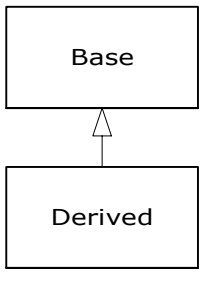
flexibility since the compiler must place compile-time restrictions on classes created with inheritance.

Because inheritance is so important in object-oriented programming it is often highly emphasized, and the new programmer can get the idea that inheritance should be used everywhere. This can result in awkward and overly complicated designs. Instead, you should first look to composition when creating new classes, since it is simpler and more flexible. If you take this approach, your designs will be cleaner. Once you've had some experience, it will be reasonably obvious when you need inheritance.

Inheritance: reusing the interface

By itself, the idea of an object is a convenient tool. It allows you to package data and functionality together by *concept*, so you can represent an appropriate problem-space idea rather than being forced to use the idioms of the underlying machine. These concepts are expressed as fundamental units in the programming language by using the **class** keyword.

It seems a pity, however, to go to all the trouble to create a class and then be forced to create a brand new one that might have similar functionality. It's nicer if we can take the existing class, clone it, and then make additions and modifications to the clone. This is effectively what you get with *inheritance*, with the exception that if the original class (called the *base* or *super* or *parent* class) is changed, the modified "clone" (called the *derived* or *inherited* or *sub* or *child* class) also reflects those changes.

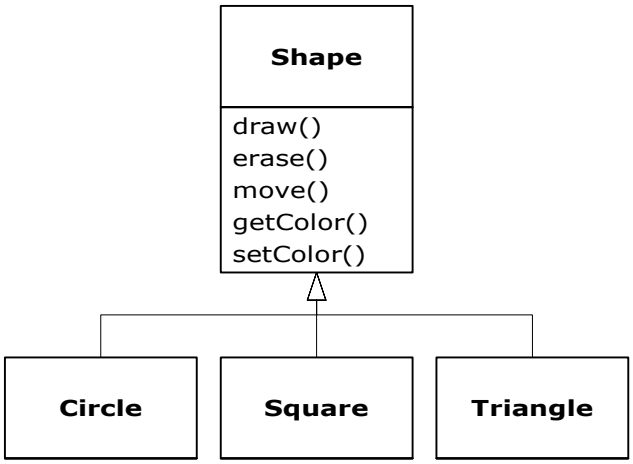


(The arrow in the above UML diagram points from the derived class to the base class. As you will see, there can be more than one derived class.)

A type does more than describe the constraints on a set of objects; it also has a relationship with other types. Two types can have characteristics and behaviors in common, but one type may contain more characteristics than another and may also handle more messages (or handle them differently). Inheritance expresses this similarity between types using the concept of base types and derived types. A base type contains all of the characteristics and behaviors that are shared among the types derived from it. You create a base type to represent the core of your ideas about some objects in your system. From the base type, you derive other types to express the different ways that this core can be realized.

For example, a trash-recycling machine sorts pieces of trash. The base type is "trash," and each piece of trash has a weight, a value, and so on, and can be shredded, melted, or decomposed. From this, more specific types of trash are derived that may have additional characteristics (a bottle has a color) or behaviors (an aluminum can may be crushed, a steel can is magnetic). In addition, some behaviors may be different (the value of paper depends on its type and condition). Using inheritance, you can build a type hierarchy that expresses the problem you're trying to solve in terms of its types.

A second example is the classic "shape" example, perhaps used in a computer-aided design system or game simulation. The base type is "shape," and each shape has a size, a color, a position, and so on. Each shape can be drawn, erased, moved, colored, etc. From this, specific types of shapes are derived (inherited): circle, square, triangle, and so on, each of which may have additional characteristics and behaviors. Certain shapes can be flipped, for example. Some behaviors may be different, such as when you want to calculate the area of a shape. The type hierarchy embodies both the similarities and differences between the shapes.



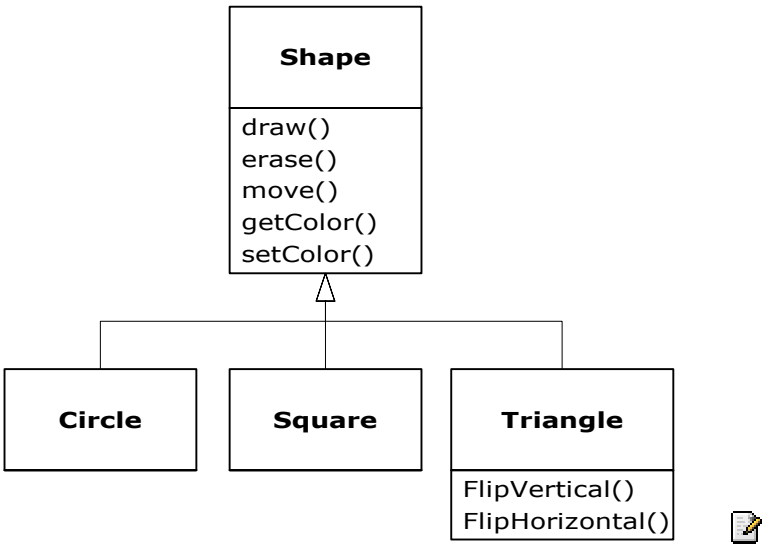
Casting the solution in the same terms as the problem is tremendously beneficial because you don't need a lot of intermediate models to get from a description of the problem to a description of the solution. With objects, the type hierarchy is the primary model, so you go directly from the description of the system in the real world to the description of the system in code. Indeed, one of the difficulties people have with object-oriented design is that it's too simple to get from the beginning to the end. A mind trained to look for complex solutions is often stumped by this simplicity at first.

When you inherit from an existing type, you create a new type. This new type contains not only all the members of the existing type (although the **private** ones are hidden away and inaccessible), but more important, it duplicates the interface of the base class. That is, all the messages you can send to objects of the base class you can also send to objects of the derived class. Since we know the type of a class by the messages we can send to it, this means that the derived class *is the same type as the base class*. In the previous example, "a circle is a shape." This type equivalence via inheritance is one of the fundamental gateways in understanding the meaning of object-oriented programming.

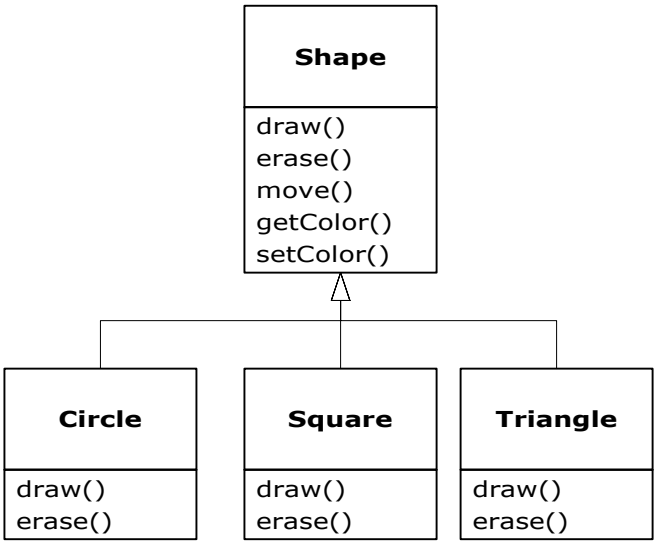
Since both the base class and derived class have the same interface, there must be some implementation to go along with that interface. That is, there must be some code to execute when an object receives a particular

message. If you simply inherit a class and don't do anything else, the methods from the base-class interface come right along into the derived class. That means objects of the derived class have not only the same type, they also have the same behavior, which isn't particularly interesting.

You have two ways to differentiate your new derived class from the original base class. The first is quite straightforward: You simply add brand new functions to the derived class. These new functions are not part of the base class interface. This means that the base class simply didn't do as much as you wanted it to, so you added more functions. This simple and primitive use for inheritance is, at times, the perfect solution to your problem. However, you should look closely for the possibility that your base class might also need these additional functions. This process of discovery and iteration of your design happens regularly in object-oriented programming.



Although inheritance may sometimes imply (especially in Java, where the keyword that indicates inheritance is **extends**) that you are going to add new functions to the interface, that's not necessarily true. The second and more important way to differentiate your new class is to *change* the behavior of an existing base-class function. This is referred to as *overriding* that function.



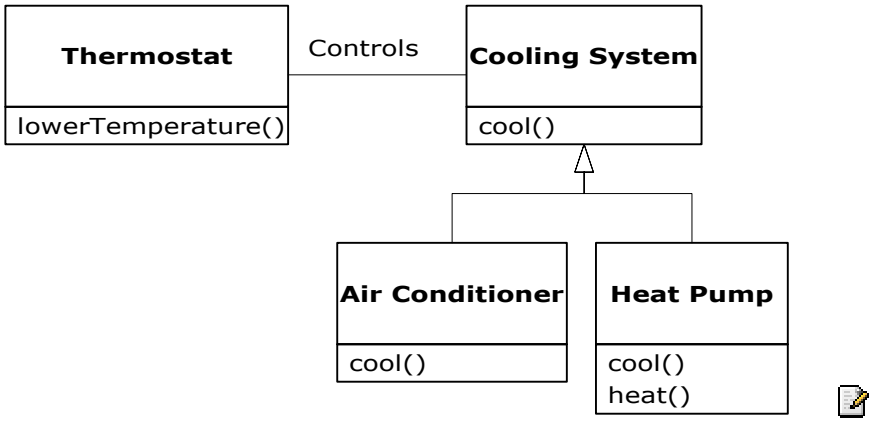
To override a function, you simply create a new definition for the function in the derived class. You're saying, "I'm using the same interface function here, but I want it to do something different for my new type."

Is-a vs. is-like-a relationships

There's a certain debate that can occur about inheritance: Should inheritance override *only* base-class functions (and not add new member functions that aren't in the base class)? This would mean that the derived type is *exactly* the same type as the base class since it has exactly the same interface. As a result, you can exactly substitute an object of the derived class for an object of the base class. This can be thought of as *pure substitution*, and it's often referred to as the *substitution principle*. In a sense, this is the ideal way to treat inheritance. We often refer to the relationship between the base class and derived classes in this case as an *is-a* relationship, because you can say "a circle *is a* shape." A test for inheritance is to determine whether you can state the is-a relationship about the classes and have it make sense.

There are times when you must add new interface elements to a derived type, thus extending the interface and creating a new type. The new type can still be substituted for the base type, but the substitution isn't perfect

because your new functions are not accessible from the base type. This can be described as an *is-like-a*[5] relationship; the new type has the interface of the old type but it also contains other functions, so you can't really say it's exactly the same. For example, consider an air conditioner. Suppose your house is wired with all the controls for cooling; that is, it has an interface that allows you to control cooling. Imagine that the air conditioner breaks down and you replace it with a heat pump, which can both heat and cool. The heat pump *is-like-an* air conditioner, but it can do more. Because the control system of your house is designed only to control cooling, it is restricted to communication with the cooling part of the new object. The interface of the new object has been extended, and the existing system doesn't know about anything except the original interface.



Of course, once you see this design it becomes clear that the base class "cooling system" is not general enough, and should be renamed to "temperature control system" so that it can also include heating—at which point the substitution principle will work. However, the diagram above is an example of what can happen in design and in the real world.

When you see the substitution principle it's easy to feel like this approach (pure substitution) is the only way to do things, and in fact it *is* nice if your design works out that way. But you'll find that there are times when

[5] My term.

it's equally clear that you must add new functions to the interface of a derived class. With inspection both cases should be reasonably obvious.

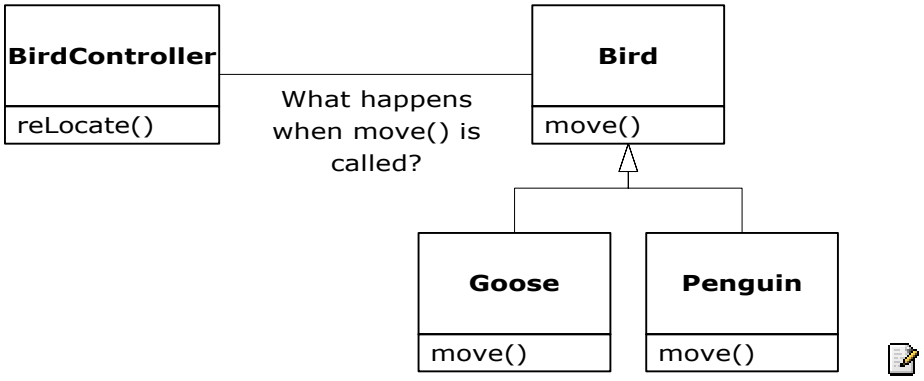
Interchangeable objects with polymorphism

When dealing with type hierarchies, you often want to treat an object not as the specific type that it is, but instead as its base type. This allows you to write code that doesn't depend on specific types. In the shape example, functions manipulate generic shapes without respect to whether they're circles, squares, triangles, or some shape that hasn't even been defined yet. All shapes can be drawn, erased, and moved, so these functions simply send a message to a shape object; they don't worry about how the object copes with the message.

Such code is unaffected by the addition of new types, and adding new types is the most common way to extend an object-oriented program to handle new situations. For example, you can derive a new subtype of shape called pentagon without modifying the functions that deal only with generic shapes. This ability to extend a program easily by deriving new subtypes is important because it greatly improves designs while reducing the cost of software maintenance.

There's a problem, however, with attempting to treat derived-type objects as their generic base types (circles as shapes, bicycles as vehicles, cormorants as birds, etc.). If a function is going to tell a generic shape to draw itself, or a generic vehicle to steer, or a generic bird to move, the compiler cannot know at compile-time precisely what piece of code will be executed. That's the whole point—when the message is sent, the programmer doesn't *want* to know what piece of code will be executed; the draw function can be applied equally to a circle, a square, or a triangle, and the object will execute the proper code depending on its specific type. If you don't have to know what piece of code will be executed, then when you add a new subtype, the code it executes can be different without requiring changes to the function call. Therefore, the compiler cannot know precisely what piece of code is executed, so what does it do? For example, in the following diagram the **BirdController** object just works

with generic **Bird** objects, and does not know what exact type they are. This is convenient from **BirdController**'s perspective because it doesn't have to write special code to determine the exact type of **Bird** it's working with, or that **Bird**'s behavior. So how does it happen that, when **move( )** is called while ignoring the specific type of **Bird**, the right behavior will occur (a **Goose** runs, flies, or swims, and a **Penguin** runs or swims)?



The answer is the primary twist in object-oriented programming: the compiler cannot make a function call in the traditional sense. The function call generated by a non-OOP compiler causes what is called *early binding*, a term you may not have heard before because you've never thought about it any other way. It means the compiler generates a call to a specific function name, and the linker resolves this call to the absolute address of the code to be executed. In OOP, the program cannot determine the address of the code until run-time, so some other scheme is necessary when a message is sent to a generic object.

To solve the problem, object-oriented languages use the concept of *late binding*. When you send a message to an object, the code being called isn't determined until run-time. The compiler does ensure that the function exists and performs type checking on the arguments and return value (a language in which this isn't true is called *weakly typed*), but it doesn't know the exact code to execute.

To perform late binding, C# uses a special bit of code in lieu of the absolute call. This code calculates the address of the function body, using information stored in the object (this process is covered in great detail in Chapter 7). Thus, each object can behave differently according to the

contents of that special bit of code. When you send a message to an object, the object actually does figure out what to do with that message.

In C#, you can choose whether a language is early- or late-bound. By default, they are early-bound. To take advantage of polymorphism, methods must be defined in the base class using the **virtual** keyword and implemented in inheriting classes with the **override** keyword.

Consider the shape example. The family of classes (all based on the same uniform interface) was diagrammed earlier in this chapter. To demonstrate polymorphism, we want to write a single piece of code that ignores the specific details of type and talks only to the base class. That code is *decoupled* from type-specific information, and thus is simpler to write and easier to understand. And, if a new type—a **Hexagon**, for example—is added through inheritance, the code you write will work just as well for the new type of **Shape** as it did on the existing types. Thus, the program is *extensible*.

If you write a method in C# (as you will soon learn how to do):

```
void DoStuff(Shape s) {
  s.Erase();
  // ...
  s.Draw();
}
```

This function speaks to any **Shape**, so it is independent of the specific type of object that it's drawing and erasing. If in some other part of the program we use the **DoStuff( )** function:

```
Circle c = new Circle();
Triangle t = new Triangle();
Line l = new Line();
DoStuff(c);
DoStuff(t);
DoStuff(l);
```

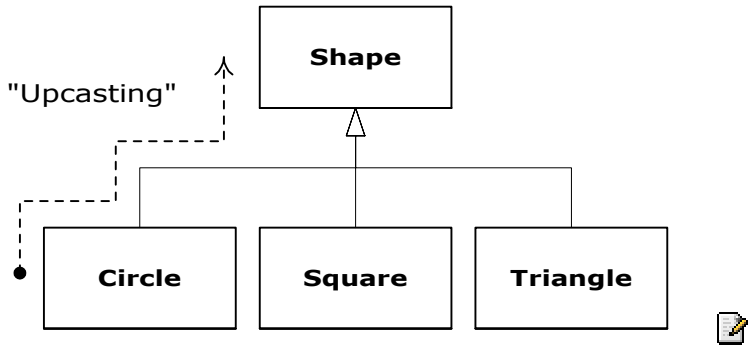
The calls to **DoStuff( )** automatically work correctly, regardless of the exact type of the object.

This is actually a pretty amazing trick. Consider the line:

```
DoStuff(c);
```

What's happening here is that a **Circle** is being passed into a function that's expecting a **Shape**. Since a **Circle** *is* a **Shape** it can be treated as one by **DoStuff( )**. That is, any message that **DoStuff( )** can send to a **Shape**, a **Circle** can accept. So it is a completely safe and logical thing to do.

We call this process of treating a derived type as though it were its base type *upcasting*. The name *cast* is used in the sense of casting into a mold and the *up* comes from the way the inheritance diagram is typically arranged, with the base type at the top and the derived classes fanning out downward. Thus, casting to a base type is moving up the inheritance diagram: "upcasting."



An object-oriented program contains some upcasting somewhere, because that's how you decouple yourself from knowing about the exact type you're working with. Look at the code in **DoStuff( )**:

```
s.Erase();
// ...
s.Draw();
```

Notice that it doesn't say "If you're a **Circle**, do this, if you're a **Square**, do that, etc." If you write that kind of code, which checks for all the possible types that a **Shape** can actually be, it's messy and you need to change it every time you add a new kind of **Shape**. Here, you just say "You're a shape, I know you can **Erase( )** and **Draw( )** yourself, do it, and take care of the details correctly."

What's impressive about the code in **DoStuff( )** is that, somehow, the right thing happens. Calling **Draw( )** for **Circle** causes different code to be executed than when calling **Draw( )** for a **Square** or a **Line**, but when the **Draw( )** message is sent to an anonymous **Shape**, the correct behavior occurs based on the actual type of the **Shape**. This is amazing because, as mentioned earlier, when the C# compiler is compiling the code for **DoStuff( )**, it cannot know exactly what types it is dealing with. So ordinarily, you'd expect it to end up calling the version of **Erase( )** and **Draw( )** for the base class **Shape**, and not for the specific **Circle**, **Square**, or **Line**. And yet the right thing happens because of polymorphism. The compiler and run-time system handle the details; all you need to know is that it happens, and more important how to design with it. When you send a message to an object, the object will do the right thing, even when upcasting is involved.

Abstract base classes and interfaces

Often in a design, you want the base class to present *only* an interface for its derived classes. That is, you don't want anyone to actually create an object of the base class, only to upcast to it so that its interface can be used. This is accomplished by making that class *abstract* using the **abstract** keyword. If anyone tries to make an object of an **abstract** class, the compiler prevents them. This is a tool to enforce a particular design.

You can also use the **abstract** keyword to describe a method that hasn't been implemented yet—as a stub indicating "here is an interface function for all types inherited from this class, but at this point I don't have any implementation for it." An **abstract** method may be created only inside an **abstract** class. When the class is inherited, that method must be implemented, or the inheriting class becomes **abstract** as well. Creating an **abstract** method allows you to put a method in an interface without being forced to provide a possibly meaningless body of code for that method.

The **interface** keyword takes the concept of an **abstract** class one step further by preventing any function definitions at all. The **interface** is a

very handy and commonly used tool, as it provides the perfect separation of interface and implementation. In addition, you can combine many interfaces together, if you wish, whereas inheriting from multiple regular classes or abstract classes is not possible.

Object landscapes and lifetimes

Technically, OOP is just about abstract data typing, inheritance, and polymorphism, but other issues can be at least as important. The remainder of this section will cover these issues.

One of the most important factors is the way objects are created and destroyed. Where is the data for an object and how is the lifetime of the object controlled? There are different philosophies at work here. C++ takes the approach that control of efficiency is the most important issue, so it gives the programmer a choice. For maximum run-time speed, the storage and lifetime can be determined while the program is being written, by placing the objects on the stack (these are sometimes called *automatic* or *scoped* variables) or in the static storage area. This places a priority on the speed of storage allocation and release, and control of these can be very valuable in some situations. However, you sacrifice flexibility because you must know the exact quantity, lifetime, and type of objects while you're writing the program. If you are trying to solve a more general problem such as computer-aided design, warehouse management, or air-traffic control, this is too restrictive.

The second approach is to create objects dynamically in a pool of memory called the heap. In this approach, you don't know until run-time how many objects you need, what their lifetime is, or what their exact type is. Those are determined at the spur of the moment while the program is running. If you need a new object, you simply make it on the heap at the point that you need it. Because the storage is managed dynamically, at run-time, the amount of time required to allocate storage on the heap is significantly longer than the time to create storage on the stack. (Creating storage on the stack is often a single assembly instruction to move the stack pointer down, and another to move it back up.) The dynamic

approach makes the generally logical assumption that objects tend to be complicated, so the extra overhead of finding storage and releasing that storage will not have an important impact on the creation of an object. In addition, the greater flexibility is essential to solve the general programming problem.

C# uses the second approach exclusively, except for *value types* which will be discussed shortly. Every time you want to create an object, you use the **new** keyword to build a dynamic instance of that object. With languages that allow objects to be created on the stack, the compiler determines how long the object lasts and can automatically destroy it. However, if you create it on the heap the compiler has no knowledge of its lifetime. In a language like C++, you must determine programmatically when to destroy the object, which can lead to memory leaks if you don't do it correctly (and this is a common problem in C++ programs). The .NET runtime provides a feature called a garbage collector that automatically discovers when an object is no longer in use and destroys it. A garbage collector is much more convenient because it reduces the number of issues that you must track and the code you must write. More important, the garbage collector provides a much higher level of insurance against the insidious problem of memory leaks (which has brought many a C++ project to its knees).

The rest of this section looks at additional factors concerning object lifetimes and landscapes.

Collections and iterators

If you don't know how many objects you're going to need to solve a particular problem, or how long they will last, you also don't know how to store those objects. How can you know how much space to create for those objects? You can't, since that information isn't known until run-time.

The solution to most problems in object-oriented design seems flippant: you create another type of object. The new type of object that solves this particular problem holds references to other objects. Of course, you can do the same thing with an array, which is available in most languages. But there's more. This new object, generally called a *container* (also called a

collection), will expand itself whenever necessary to accommodate everything you place inside it. So you don't need to know how many objects you're going to hold in a container. Just create a container object and let it take care of the details.

Fortunately, a good OOP language comes with a set of containers as part of the package. In C++, it's part of the Standard C++ Library and is sometimes called the Standard Template Library (STL). Object Pascal has containers in its Visual Component Library (VCL). Smalltalk has a very complete set of containers. C# also has containers in its standard library. In some libraries, a generic container is considered good enough for all needs, and in others (C#, for example) the library has different types of containers for different needs: a vector (called an **ArrayList** in C#) for consistent access to all elements, queues, hashtables, trees, stacks, etc.

All containers have some way to put things in and get things out; there are usually functions to add elements to a container, and others to fetch those elements back out. But fetching elements can be more problematic, because a single-selection function is restrictive. What if you want to manipulate or compare a set of elements in the container instead of just one?

The solution is an iterator, which is an object whose job is to select the elements within a container and present them to the user of the iterator. As a class, it also provides a level of abstraction. This abstraction can be used to separate the details of the container from the code that's accessing that container. The container, via the iterator, is abstracted to be simply a sequence. The iterator allows you to traverse that sequence without worrying about the underlying structure—that is, whether it's an **ArrayList**, a **Hashtable**, a **Stack**, or something else. This gives you the flexibility to easily change the underlying data structure without disturbing the code in your program.

From a design standpoint, all you really want is a sequence that can be manipulated to solve your problem. If a single type of sequence satisfied all of your needs, there'd be no reason to have different kinds. There are two reasons that you need a choice of containers. First, containers provide different types of interfaces and external behavior. A stack has a different interface and behavior than that of a queue, which is different from that of

a dictionary or a list. One of these might provide a more flexible solution to your problem than the other. Second, different containers have different efficiencies for certain operations. But in the end, remember that a container is only a storage cabinet to put objects in. If that cabinet solves all of your needs, it doesn't really matter how it is implemented (a basic concept with most types of objects).

The singly rooted hierarchy

One of the issues in OOP that has become especially prominent since the introduction of C++ is whether all classes should ultimately be inherited from a single base class. In C# (as with virtually all other OOP languages) the answer is "yes" and the name of this ultimate base class is simply **object**. It turns out that the benefits of the singly rooted hierarchy are many.

All objects in a singly rooted hierarchy have an interface in common, so they are all ultimately the same type. The alternative (provided by C++) is that you don't know that everything is the same fundamental type. From a backward-compatibility standpoint this fits the model of C better and can be thought of as less restrictive, but when you want to do full-on object-oriented programming you must then build your own hierarchy to provide the same convenience that's built into other OOP languages. And in any new class library you acquire, some other incompatible interface will be used. It requires effort (and possibly multiple inheritance) to work the new interface into your design. Is the extra "flexibility" of C++ worth it? If you need it—if you have a large investment in C—it's quite valuable. If you're starting from scratch, other alternatives such as C# can often be more productive.

All objects in a singly rooted hierarchy (such as C# provides) can be guaranteed to have certain functionality. You know you can perform certain basic operations on every object in your system. A singly rooted hierarchy, along with creating all objects on the heap, greatly simplifies argument passing (one of the more complex topics in C++).

A singly rooted hierarchy makes it much easier to implement a garbage collector (which is conveniently built into C#). The necessary support can be installed in the base class, and the garbage collector can thus send the

appropriate messages to every object in the system. Without a singly rooted hierarchy and a system to manipulate an object via a reference, it is difficult to implement a garbage collector.

Since run-time type information is guaranteed to be in all objects, you'll never end up with an object whose type you cannot determine. This is especially important with system level operations, such as exception handling, and to allow greater flexibility in programming.

Collection libraries and support for easy collection use

Because a container is a tool that you'll use frequently, it makes sense to have a library of containers that are built in a reusable fashion, so you can take one off the shelf and plug it into your program. .NET provides such a library, which should satisfy most needs.

Downcasting vs. templates/generics

To make these containers reusable, they hold the one universal type in .NET that was previously mentioned: **object**. The singly rooted hierarchy means that everything is an **object**, so a container that holds **object**s can hold anything. This makes containers easy to reuse.

To use such a container, you simply add object references to it, and later ask for them back. But, since the container holds only **object**s, when you add your object reference into the container it is upcast to **object**, thus losing its identity. When you fetch it back, you get an **object** reference, and not a reference to the type that you put in. So how do you turn it back into something that has the useful interface of the object that you put into the container?

Here, the cast is used again, but this time you're not casting up the inheritance hierarchy to a more general type, you cast down the hierarchy to a more specific type. This manner of casting is called downcasting. With upcasting, you know, for example, that a **Circle** is a type of **Shape** so it's safe to upcast, but you don't know that an **object** is necessarily a **Circle** or a **Shape** so it's hardly safe to downcast unless you know that's what you're dealing with.

It's not completely dangerous, however, because if you downcast to the wrong thing you'll get a run-time error called an *exception,* which will be described shortly. When you fetch object references from a container, though, you must have some way to remember exactly what they are so you can perform a proper downcast.

Downcasting and the run-time checks require extra time for the running program, and extra effort from the programmer. Wouldn't it make sense to somehow create the container so that it knows the types that it holds, eliminating the need for the downcast and a possible mistake? The solution is parameterized types, which are classes that the compiler can automatically customize to work with particular types. For example, with a parameterized container, the compiler could customize that container so that it would accept only Shapes and fetch only Shapes.

Parameterized types are an important part of C++, partly because C++ has no singly rooted hierarchy. In C++, the keyword that implements parameterized types is "template." .NET currently has no parameterized types since it is possible for it to get by—however awkwardly—using the singly rooted hierarchy. However, there is no doubt that parameterized types will be implemented in a future version of the .NET Framework.

The housekeeping dilemma: who should clean up?

Each object requires resources in order to exist, most notably memory. When an object is no longer needed it must be cleaned up so that these resources are released for reuse. In simple programming situations the question of how an object is cleaned up doesn't seem too challenging: you create the object, use it for as long as it's needed, and then it should be destroyed. It's not hard, however, to encounter situations in which the situation is more complex.

Suppose, for example, you are designing a system to manage air traffic for an airport. (The same model might also work for managing crates in a warehouse, or a video rental system, or a kennel for boarding pets.) At first it seems simple: make a container to hold airplanes, then create a new airplane and place it in the container for each airplane that enters the

air-traffic-control zone. For cleanup, simply delete the appropriate airplane object when a plane leaves the zone.

But perhaps you have some other system to record data about the planes; perhaps data that doesn't require such immediate attention as the main controller function. Maybe it's a record of the flight plans of all the small planes that leave the airport. So you have a second container of small planes, and whenever you create a plane object you also put it in this second container if it's a small plane. Then some background process performs operations on the objects in this container during idle moments.

Now the problem is more difficult: how can you possibly know when to destroy the objects? When you're done with the object, some other part of the system might not be. This same problem can arise in a number of other situations, and in programming systems (such as C++) in which you must explicitly delete an object when you're done with it this can become quite complex.

With C#, the garbage collector is designed to take care of the problem of releasing the memory (although this doesn't include other aspects of cleaning up an object). The garbage collector "knows" when an object is no longer in use, and it then automatically releases the memory for that object. This (combined with the fact that all objects are inherited from the single root class **object** and that you can create objects only one way, on the heap) makes the process of programming in C# much simpler than programming in C++. You have far fewer decisions to make and hurdles to overcome.

Garbage collectors vs. efficiency and flexibility

If all this is such a good idea, why didn't they do the same thing in C++? Well of course there's a price you pay for all this programming convenience, and that price is run-time overhead. As mentioned before, in C++ you can create objects on the stack, and in this case they're automatically cleaned up (but you don't have the flexibility of creating as many as you want at run-time). Creating objects on the stack is the most efficient way to allocate storage for objects and to free that storage.

Creating objects on the heap can be much more expensive. Always inheriting from a base class and making all function calls polymorphic also exacts a small toll. But the garbage collector is a particular problem because you never quite know when it's going to start up or how long it will take. This means that there's an inconsistency in the rate of execution of a C# program, so you can't use it in certain situations, such as when the rate of execution of a program is uniformly critical. (These are generally called real time programs, although not all real time programming problems are this stringent.)

The designers of the C++ language, trying to woo C programmers (and most successfully, at that), did not want to add any features to the language that would impact the speed or the use of C++ in any situation where programmers might otherwise choose C. This goal was realized, but at the price of greater complexity when programming in C++. C# is simpler than C++, but the trade-off is in efficiency and sometimes applicability. For a significant portion of programming problems, however, C# is the superior choice.

Exception handling: dealing with errors

Ever since the beginning of programming languages, error handling has been one of the most difficult issues. Because it's so hard to design a good error handling scheme, many languages simply ignore the issue, passing the problem on to library designers who come up with halfway measures that can work in many situations but can easily be circumvented, generally by just ignoring them. A major problem with most error handling schemes is that they rely on programmer vigilance in following an agreed-upon convention that is not enforced by the language. If the programmer is not vigilant—often the case if they are in a hurry—these schemes can easily be forgotten.

Exception handling wires error handling directly into the programming language and sometimes even the operating system. An exception is an object that is "thrown" from the site of the error and can be "caught" by an appropriate exception handler designed to handle that particular type of

error. It's as if exception handling is a different, parallel path of execution that can be taken when things go wrong. And because it uses a separate execution path, it doesn't need to interfere with your normally executing code. This makes that code simpler to write since you aren't constantly forced to check for errors. In addition, a thrown exception is unlike an error value that's returned from a function or a flag that's set by a function in order to indicate an error condition—these can be ignored. An exception cannot be ignored, so it's guaranteed to be dealt with at some point. Finally, exceptions provide a way to reliably recover from a bad situation. Instead of just exiting you are often able to set things right and restore the execution of a program, which produces much more robust programs.

It's worth noting that exception handling isn't an object-oriented feature, although in object-oriented languages the exception is normally represented with an object. Exception handling existed before object-oriented languages.

Multithreading

A fundamental concept in computer programming is the idea of handling more than one task at a time. Many programming problems require that the program be able to stop what it's doing, deal with some other problem, and then return to the main process. The solution has been approached in many ways. Initially, programmers with low-level knowledge of the machine wrote interrupt service routines and the suspension of the main process was initiated through a hardware interrupt. Although this worked well, it was difficult and nonportable, so it made moving a program to a new type of machine slow and expensive.

Sometimes interrupts are necessary for handling time-critical tasks, but there's a large class of problems in which you're simply trying to partition the problem into separately running pieces so that the whole program can be more responsive. Within a program, these separately running pieces are called threads, and the general concept is called *multithreading*. A common example of multithreading is the user interface. By using

threads, a user can press a button and get a quick response rather than being forced to wait until the program finishes its current task.

Ordinarily, threads are just a way to allocate the time of a single processor. But if the operating system supports multiple processors, each thread can be assigned to a different processor and they can truly run in parallel. One of the convenient features of multithreading at the language level is that the programmer doesn't need to worry about whether there are many processors or just one. The program is logically divided into threads and if the machine has more than one processor then the program runs faster, without any special adjustments.

All this makes threading sound pretty simple. There is a catch: shared resources. If you have more than one thread running that's expecting to access the same resource you have a problem. For example, two processes can't simultaneously send information to a printer. To solve the problem, resources that can be shared, such as the printer, must be locked while they are being used. So a thread locks a resource, completes its task, and then releases the lock so that someone else can use the resource.

C#'s threading is built into the language, which makes a complicated subject much simpler. The threading is supported on an object level, so one thread of execution is represented by one object. C# also provides limited resource locking. It can lock the memory of any object (which is, after all, one kind of shared resource) so that only one thread can use it at a time. This is accomplished with the **lock** keyword. Other types of resources must be locked explicitly by the programmer, typically by creating an object to represent the lock that all threads must check before accessing that resource.

Persistence

tk

C# and the Internet

tk

What is the Web?

tk

Client/Server computing

tk

The Web as a giant server

tk

Client-side programming

tk

Plug-ins

tk

Scripting languages

tk

Rich clients in C#

tk

Security

tk

Internet vs. intranet

tk

Server-side programming

tk

A separate arena: applications

tk

Analysis and design

tk

Extreme programming

tk

Why .NET succeeds

Systems are easier to express and understand

Maximal leverage with libraries

Error handling

Error handling in C is a notorious problem, and one that is often ignored—finger-crossing is usually involved. If you're building a large, complex program, there's nothing worse than having an error buried somewhere with no clue as to where it came from. Java *exception handling* is a way to guarantee that an error is noticed, and that something happens as a result.

Programming in the large

Many traditional languages have built-in limitations to program size and complexity. BASIC, for example, can be great for pulling together quick solutions for certain classes of problems, but if the program gets more than a few pages long, or ventures out of the normal problem domain of that language, it's like trying to swim through an ever-more viscous fluid. There's no clear line that tells you when your language is failing you, and even if there were, you'd ignore it. You don't say, "My BASIC program just got too big; I'll have to rewrite it in C!" Instead, you try to shoehorn a few more lines in to add that one new feature. So the extra costs come creeping up on you.

C# is designed to aid *programming in the large*—that is, to erase those creeping-complexity boundaries between a small program and a large one. You certainly don't need to use OOP when you're writing a "hello world" style utility program, but the features are there when you need them. And the compiler is aggressive about ferreting out bug-producing errors for small and large programs alike.

Strategies for transition

tk

Guidelines

Here are some guidelines to consider when making the transition to .NET and C#:

1. Training

The first step is some form of education. Remember the company's investment in code, and try not to throw everything into disarray for six to nine months while everyone puzzles over how interfaces work. Pick a small group for indoctrination, preferably one composed of people who are curious, work well together, and can function as their own support network while they're learning C# and .NET.

An alternative approach that is sometimes suggested is the education of all company levels at once, including overview courses for strategic managers as well as design and programming courses for project builders. This is especially good for smaller companies making fundamental shifts in the way they do things, or at the division level of larger companies. Because the cost is higher, however, some may choose to start with project-level training, do a pilot project (possibly with an outside mentor), and let the project team become the teachers for the rest of the company.

2. Low-risk project

Try a low-risk project first and allow for mistakes. Once you've gained some experience, you can either seed other projects from members of this first team or use the team members as a .NET technical support staff. This

first project may not work right the first time, so it should not be mission-critical for the company. It should be simple, self-contained, and instructive; this means that it should involve creating classes that will be meaningful to the other programmers in the company when they get their turn to learn C# and .NET.

3. Model from success

Seek out examples of good object-oriented design before starting from scratch. There's a good probability that someone has solved your problem already, and if they haven't solved it exactly you can probably apply what you've learned about abstraction to modify an existing design to fit your needs. This is the general concept of *design patterns,* covered in *Thinking in Patterns with Java*, downloadable at *www.BruceEckel.com*.

4. Use existing class libraries

The primary economic motivation for switching to OOP is the easy use of existing code in the form of class libraries (in particular, the .NET Framework SDK libraries, which are covered throughout this book). The shortest application development cycle will result when you can create and use objects from off-the-shelf libraries. However, some new programmers don't understand this, are unaware of existing class libraries, or, through fascination with the language, desire to write classes that may already exist. Your success with OOP, .NET, and C# will be optimized if you make an effort to seek out and reuse other people's code early in the transition process.

5. Don't rewrite existing code in C#

It is not usually the best use of your time to take existing, functional code and rewrite it in C#. (If you must turn it into objects, you can interface to the C or C++ code as discussed in #ref#.) There are incremental benefits, especially if the code is slated for reuse. But chances are you aren't going to see the dramatic increases in productivity that you hope for in your first few projects unless that project is a new one. C# and .NET shine best when taking a project from concept to reality.

Management obstacles

If you're a manager, your job is to acquire resources for your team, to overcome barriers to your team's success, and in general to try to provide the most productive and enjoyable environment so your team is most likely to perform those miracles that are always being asked of you. Moving to .NET falls in all three of these categories, and it would be wonderful if it didn't cost you anything as well. Although moving to .NET may be cheaper—depending on your constraints—than the alternatives for a team of Visual Basic programmers (and probably for programmers in other procedural languages), it isn't free, and there are obstacles you should be aware of before trying to sell the move to C# within your company and embarking on the move itself.

Startup costs

tk

Performance issues

tk

Common design errors

tk

C# vs. Java?

tk

Summary

tk

3: Hello, Objects

Although it is based on C++, C# is more of a "pure" object-oriented language.

Both C++ and C# are hybrid languages, but in C# the designers felt that the hybridization was not as important as it was in C++. A hybrid language allows multiple programming styles; the reason C++ is hybrid is to support backward compatibility with the C language. Because C++ is a superset of the C language, it includes many of that language's undesirable features, which can make some aspects of C++ overly complicated.

The C# language assumes that you want to do only object-oriented programming. This means that before you can begin you must shift your mindset into an object-oriented world (unless it's already there). The benefit of this initial effort is the ability to program in a language that is simpler to learn and to use than many other OOP languages. In this chapter we'll see the basic components of a C# program and we'll learn that everything in C# is an object, even a C# program.

You manipulate objects with references

Each programming language has its own means of manipulating data. Sometimes the programmer must be constantly aware of what type of manipulation is going on. Are you manipulating the object directly, or are you dealing with some kind of indirect representation (a pointer in C or C++) that must be treated with a special syntax?

All this is simplified in C#. You treat everything as an object, so there is a single consistent syntax that you use everywhere. Although you *treat* everything as an object, the identifier you manipulate is actually a

"reference" to an object. You might imagine this scene as a television (the object) with your remote control (the reference). As long as you're holding this reference, you have a connection to the television, but when someone says "change the channel" or "lower the volume," what you're manipulating is the reference, which in turn modifies the object. If you want to move around the room and still control the television, you take the remote/reference with you, not the television.

Also, the remote control can stand on its own, with no television. That is, just because you have a reference doesn't mean there's necessarily an object connected to it. So if you want to hold a word or sentence, you create a **string** reference:

```
string s;
```

But here you've created *only* the reference, not an object. If you decided to send a message to **s** at this point, you'll get an error (at run-time) because **s** isn't actually attached to anything (there's no television). A safer practice, then, is always to initialize a reference when you create it:

```
string s = "asdf";
```

However, this uses a special feature: strings can be initialized with quoted text. Normally, you must use a more general type of initialization for objects.

You must create all the objects

When you create a reference, you want to connect it with a new object. You do so, in general, with the **new** keyword. **new** says, "Make me a new one of these objects." So in the above example, you can say:

```
string s = new string("asdf");
```

Not only does this mean "Make me a new **string**," but it also gives information about *how* to make the **string** by supplying an initial character string.

Of course, **string** is not the only type that exists. C# comes with a plethora of ready-made types. What's more important is that you can create your own types. In fact, that's the fundamental activity in C# programming, and it's what you'll be learning about in the rest of this book.

Where storage lives

It's useful to visualize some aspects of how things are laid out while the program is running, in particular how memory is arranged. There are six different places to store data:

1. **Registers**. This is the fastest storage because it exists in a place different from that of other storage: inside the processor. However, the number of registers is severely limited, so registers are allocated by the compiler according to its needs. You don't have direct control, nor do you see any evidence in your programs that registers even exist.

2. **The stack**. This lives in the general RAM (random-access memory) area, but has direct support from the processor via its *stack pointer*. The stack pointer is moved down to create new memory and moved up to release that memory. This is an extremely fast and efficient way to allocate storage, second only to registers. The C# just-in-time compiler must know, while it is creating the program, the exact size and lifetime of all the data that is stored on the stack, because it must generate the code to move the stack pointer up and down. This constraint places limits on the flexibility of your programs, so while some C# storage exists on the stack—in particular, object references—C# objects themselves are not placed on the stack.

3. **The heap**. This is a general-purpose pool of memory (also in the RAM area) where all C# objects live. The nice thing about the heap is that, unlike the stack, the compiler doesn't need to know how much storage it needs to allocate from the heap or how long that storage must stay on the heap. Thus, there's a great deal of flexibility in using storage on the heap. Whenever you need to create an object, you simply write the code to create it using **new**,

and the storage is allocated on the heap when that code is executed. Of course there's a price you pay for this flexibility: it takes more time to allocate heap storage than it does to allocate stack storage (that is, if you even *could* create objects on the stack in C#, as you can in C++).

4. **Static storage**. "Static" is used here in the sense of "in a fixed location" (although it's also in RAM). Static storage contains data that is available for the entire time a program is running. You can use the **static** keyword to specify that a particular element of an object is static, but C# objects themselves are never placed in static storage.

5. **Constant storage**. Constant values are often placed directly in the program code, which is safe since they can never change. Sometimes constants are cordoned off by themselves so that they can be optionally placed in read-only memory (ROM).

6. **Non-RAM storage**. If data lives completely outside a program it can exist while the program is not running, outside the control of the program. The two primary examples of this are *streamed objects,* in which objects are turned into streams of bytes, generally to be sent to another machine, and *persistent objects,* in which the objects are placed on disk so they will hold their state even when the program is terminated. The trick with these types of storage is turning the objects into something that can exist on the other medium, and yet can be resurrected into a regular RAM-based object when necessary. C# provides support for *lightweight persistence*, and future versions of .NET might provide more complete solutions for persistence.

Arrays in Java

Virtually all programming languages support arrays. Using arrays in C and C++ is perilous because those arrays are only blocks of memory. If a program accesses the array outside of its memory block or uses the memory before initialization (common programming errors) there will be unpredictable results.

One of the primary goals of C# is safety, so many of the problems that plague programmers in C and C++ are not repeated in C#. A C# array is guaranteed to be initialized and cannot be accessed outside of its range. The range checking comes at the price of having a small amount of memory overhead on each array as well as verifying the index at run-time, but the assumption is that the safety and increased productivity is worth the expense.

When you create an array of objects, you are really creating an array of references, and each of those references is automatically initialized to a special value with its own keyword: **null**. When C# sees **null**, it recognizes that the reference in question isn't pointing to an object. You must assign an object to each reference before you use it, and if you try to use a reference that's still **null,** the problem will be reported at run-time. Thus, typical array errors are prevented in C#.

You can also create an array of value types. Again, the compiler guarantees initialization because it zeroes the memory for that array.

Arrays will be covered in detail in later chapters.

Special case: value types

Unlike "pure" object-oriented languages such as Smalltalk, C# does not insist that every variable must be an object. While the performance of most systems is not determined by a single factor, the allocation of many small objects can be notoriously costly. A story goes that in the early 1990s, a manager decreed that his programming team switch to Smalltalk to gain the benefits of object-orientation; an obstinate C programmer immediately ported the application's core matrix-manipulating algorithm to Smalltalk. The manager was pleased with this surprisingly cooperative behavior, as the programmer made sure that everyone knew that he was integrating the new Smalltalk code that very afternoon and running it through the stress test before making it the first Smalltalk code to be integrated into the production code. Twenty-four hours later, when the matrix manipulation had not completed, the manager realized that he'd been had, and never spoke of Smalltalk again.

When Java became popular, many people predicted similar performance problems. However, Java has "primitive" types for integers and characters

and so forth and many people have found that this has been sufficient to make Java appropriate for almost all performance-oriented tasks. C# goes a step beyond; not only are values (rather than classes) used for basic numeric types, developers can create new value types in the form of enumerations (**enum**s) and structures (**struct**s).

Value types can be transparently converted to object references via a process known as "boxing." This is a nice advantage of C# over Java, where turning a primitive type into an object reference requires an explicit method call.

You never need to destroy an object

In most programming languages, the concept of the lifetime of a variable occupies a significant portion of the programming effort. How long does the variable last? If you are supposed to destroy it, when should you? Confusion over variable lifetimes can lead to a lot of bugs, and this section shows how C# greatly simplifies the issue by doing all the cleanup work for you.

Scoping

Most procedural languages have the concept of *scope*. This determines both the visibility and lifetime of the names defined within that scope. In C, C++, and C#, scope is determined by the placement of curly braces **{}**. So for example:

```
{
  int x = 12;
  /* only x available */
  {
    int q = 96;
    /* both x & q available */
  }
  /* only x available */
  /* q "out of scope" */
}
```

A variable defined within a scope is available only to the end of that scope.

Indentation makes C# code easier to read. Since C# is a free-form language, the extra spaces, tabs, and carriage returns do not affect the resulting program.

Note that you *cannot* do the following, even though it is legal in C and C++:

```
{
  int x = 12;
  {
    int x = 96; /* illegal */
  }
}
```

The compiler will announce that the variable **x** has already been defined. Thus the C and C++ ability to "hide" a variable in a larger scope is not allowed because the C# designers thought that it led to confusing programs.

Scope of objects

C# objects do not have the same lifetimes as value types. When you create a C# object using **new**, it hangs around past the end of the scope. Thus if you use:

```
{
  string s = new string("a string");
} /* end of scope */
```

the reference **s** vanishes at the end of the scope. However, the **string** object that **s** was pointing to is still occupying memory. In this bit of code, there is no way to access the object because the only reference to it is out of scope. In later chapters you'll see how the reference to the object can be passed around and duplicated during the course of a program.

It turns out that because objects created with **new** stay around for as long as you want them, a whole slew of C++ programming problems simply vanish in C#. The hardest problems seem to occur in C++ because you don't get any help from the language in making sure that the objects are

available when they're needed. And more important, in C++ you must make sure that you destroy the objects when you're done with them.

That brings up an interesting question. If C# leaves the objects lying around, what keeps them from filling up memory and halting your program? This is exactly the kind of problem that would occur in C++. This is where a bit of magic happens. The .NET runtime has a *garbage collector*, which looks at all the objects that were created with **new** and figures out which ones are not being referenced anymore. Then it releases the memory for those objects, so the memory can be used for new objects. This means that you never need to worry about reclaiming memory yourself. You simply create objects, and when you no longer need them they will go away by themselves. This eliminates a certain class of programming problem: the so-called "memory leak," in which a programmer forgets to release memory.

Creating new data types: class

If everything is an object, what determines how a particular class of object looks and behaves? Put another way, what establishes the *type* of an object? You might expect there to be a keyword called "type," and that certainly would have made sense. Historically, however, most object-oriented languages have used the keyword **class** to mean "I'm about to tell you what a new type of object looks like." The **class** keyword (which is so common that it will not be emboldened throughout this book) is followed by the name of the new type. For example:

```
class ATypeName { /* class body goes here */ }
```

This introduces a new type, so you can now create an object of this type using **new**:

```
ATypeName a = new ATypeName();
```

In **ATypeName**, the class body consists only of a comment (the stars and slashes and what is inside, which will be discussed later in this chapter), so there is not too much that you can do with it. In fact, you cannot tell it

to do much of anything (that is, you cannot send it any interesting messages) until you define some methods for it.

Fields, Properties, and methods

When you define a class, you can put two types of elements in your class: data members (sometimes called *fields*), member functions (typically called *methods*), and properties. A data member is an object of any type that you can communicate with via its reference. It can also be a value type (which isn't a reference). If it is a reference to an object, you must initialize that reference to connect it to an actual object (using **new**, as seen earlier) in a special function called a *constructor* (described fully in Chapter 4). If it is a primitive type you can initialize it directly at the point of definition in the class. (As you'll see later, references can also be initialized at the point of definition.)

Each object keeps its own storage for its data members; the data members are not shared among objects. Here is an example of a class with some data members:

```
class DataOnly {
  int i;
  float f;
  boolean b;
}
```

This class doesn't *do* anything, but you can create an object:

```
DataOnly d = new DataOnly();
```

You can assign values to the data members, but you must first know how to refer to a member of an object. This is accomplished by stating the name of the object reference, followed by a period (dot), followed by the name of the member inside the object:

```
objectReference.member
```

For example:

```
d.i = 47;
d.f = 1.1f;
d.b = false;
```

It is also possible that your object might contain other objects that contain data you'd like to modify. For this, you just keep "connecting the dots." For example:

```
myPlane.leftTank.capacity = 100;
```

The **DataOnly** class cannot do much of anything except hold data, because it has no member functions (methods). To understand how those work, you must first understand *arguments* and *return values*, which will be described shortly.

Default values for primitive members

When a primitive data type is a member of a class, it is guaranteed to get a default value if you do not initialize it:

Primitive type	Default
boolean	**false**
char	**'\u0000' (null)**
byte	**(byte)0**
short	**(short)0**
int	**0**
long	**0L**
float	**0.0f**
double	**0.0d**

Note carefully that the default values are what C# guarantees when the variable is used *as a member of a class*. This ensures that member variables of primitive types will always be initialized (something C++ doesn't do), reducing a source of bugs. However, this initial value may not be correct or even legal for the program you are writing. It's best to always explicitly initialize your variables.

This guarantee doesn't apply to "local" variables—those that are not fields of a class. Thus, if within a function definition you have:

```
int x;
```

Then **x** will get some arbitrary value (as in C and C++); it will not automatically be initialized to zero. You are responsible for assigning an appropriate value before you use **x**. If you forget, C# definitely improves on C++: you get a compile-time error telling you the variable might not have been initialized. (Many C++ compilers will warn you about uninitialized variables, but in C# these are errors.)

Methods, arguments, and return values

Up until now, the term *function* has been used to describe a named subroutine. The term that is more commonly used in C# is *method,* as in "a way to do something." If you want, you can continue thinking in terms of functions. It's really only a syntactic difference, but from now on "method" will be used in this book rather than "function."

Methods in C# determine the messages an object can receive. In this section you will learn how simple it is to define a method.

The fundamental parts of a method are the name, the arguments, the return type, and the body. Here is the basic form:

```
returnType MethodName( /* Argument list */ ) {
  /* Method body */
}
```

The return type is the type of the value that pops out of the method after you call it. The argument list gives the types and names for the information you want to pass into the method. The method name and argument list together uniquely identify the method.

Methods in C# can be created only as part of a class. A method can be called only for an object,[1] and that object must be able to perform that method call. If you try to call the wrong method for an object, you'll get an

[1] **static** methods, which you'll learn about soon, can be called *for the class*, without an object.

error message at compile-time. You call a method for an object by naming the object followed by a period (dot), followed by the name of the method and its argument list, like this: **objectName.MethodName(arg1, arg2, arg3)**. For example, suppose you have a method **F( )** that takes no arguments and returns a value of type **int**. Then, if you have an object called **a** for which **F( )** can be called, you can say this:

```
int x = a.F();
```

The type of the return value must be compatible with the type of **x**.

This act of calling a method is commonly referred to as *sending a message to an object*. In the above example, the message is **F( )** and the object is **a**. Object-oriented programming is often summarized as simply "sending messages to objects."

The argument list

The method argument list specifies what information you pass into the method. As you might guess, this information—like everything else in C#—takes the form of objects. So, what you must specify in the argument list are the types of the objects to pass in and the name to use for each one. As in any situation in C# where you seem to be handing objects around, you are actually passing references[2]. The type of the reference must be correct, however. If the argument is supposed to be a **string**, what you pass in must be a string.

Consider a method that takes a **string** as its argument. Here is the definition, which must be placed within a class definition for it to be compiled:

```
int storage(string s) {
  return s.Length * 2;
}
```

This method tells you how many bytes are required to hold the information in a particular **string.** (Each **char** in a **string** is 16 bits, or two bytes, long, to support Unicode characters. Check this – doesn't it

[2] The C# keyword **ref** allows you to override this behavior

have more to do with the TextEncoding?) The argument is of type **string** and is called **s**. Once **s** is passed into the method, you can treat it just like any other object. (You can send messages to it.) Here, the **Length** property is used, which is one of the properties of **string**s; it returns the number of characters in a string.

You can also see the use of the **return** keyword, which does two things. First, it means "leave the method, I'm done." Second, if the method produces a value, that value is placed right after the **return** statement. In this case, the return value is produced by evaluating the expression **s.Length * 2**.

You can return any type you want, but if you don't want to return anything at all, you do so by indicating that the method returns **void**. Here are some examples:

```
boolean Flag() { return true; }
float NaturalLogBase() { return 2.718f; }
void Nothing() { return; }
void Nothing2() {}
```

When the return type is **void**, then the **return** keyword is used only to exit the method, and is therefore unnecessary when you reach the end of the method. You can return from a method at any point, but if you've given a non-**void** return type then the compiler will force you (with error messages) to return the appropriate type of value regardless of where you return.

At this point, it can look like a program is just a bunch of objects with methods that take other objects as arguments and send messages to those other objects. That is indeed much of what goes on, but in the following chapter you'll learn how to do the detailed low-level work by making decisions within a method. For this chapter, sending messages will suffice.

Attributes and Meta-Behavior

The most intriguing low-level feature of the .NET Runtime is the attribute, which allows you to specify arbitrary meta-information to be associated with code elements such as classes, types, and methods. Attributes are specified in C# using square brackets just before the code element. Adding an attribute to a code element doesn't change the behavior of the code element; rather, programs can be written which say "For all the code elements that have this attribute, do this behavior." The most immediately powerful demonstration of this is the **[WebMethod]** attribute which within Visual Studio .NET is all that is necessary to trigger the exposure of that method as a Web Service.

Attributes can be used to simply tag a code element, as with **[WebMethod]** or they can contain parameters that contain additional information. For instance, this example shows an **XMLElement** attribute that specifies that, when serialized to an XML document, the **FlightSegment[ ]** array should be created as a series of individual **FlightSegment** elements:

```
[XmlElement(
   ElementName = "FlightSegment")]
   public FlightSegment[] flights;
```

Delegates

In addition to classes and value types, C# has an object-oriented type that specifies a method *signature*. A method's signature consists of its argument list and its return type. A **delegate** is a type that allows any method whose signature is identical to that specified in the delegate definition to be used as an "instance" of that delegate. In this way, a method can be used as if it were a variable – instantiated, assigned to, passed around in reference form, etc.

In this example, a delegate named Foo is defined:

```
delegate void Foo(int x);

class MyClass{
   void SomeMethod(int y){ … }
   int AnotherMethod(int z){ … }
}
class MyOtherClass{
   void YetAnother(){ … }
   static void SomeOther(int z){ … }
}
```

The method **MyClass.SomeMethod( )** could be used to instantiate the **Foo** delegate, as could the method **MyOtherClass.SomeOther( )**. Neither **MyClass.AnotherMethod( )** nor **MyOtherClass.YetAnother( )** can be used as **Foo**s, as they have different signatures.

Instantiating a reference to a delegate is just like making a reference to a class:

```
Foo f = MyOtherClass.SomeOther;
```

The left-hand size contains a declaration of a variable **f** of type delegate **Foo**. The right-hand size *specifies* a method, it does not actually *call* the method **SomeOther( )**.

Delegates are a major element in programming Windows Forms, but they represent a major design feature in C# and are useful in many situations.

Properties

Fields should, essentially, never be available directly to the outside world. Mistakes are often made when a field is assigned to; the field is supposed to store a distance in metric not English units, strings are supposed to be all lowercase, etc. However, such mistakes are often not *found* until the field is used at a much later time (like, say, when preparing to enter Mars orbit). While such logical mistakes cannot be discovered by any automatic means, discovering them can be made easier by only allowing fields to be accessed via methods (which, in turn, can provide additional sanity checks and logging traces).

C# allows you to give your classes the appearance of having fields directly exposed but in fact hiding them behind method invocations. These **Property** fields come in two varieties: read-only fields that cannot be assigned to, and the more common read-and-write fields. Additionally, properties allow you to use a different type internally to store the data from the type you expose. For instance, you might wish to expose a field as an easy-to-use **bool**, but store it internally within an efficient **BitArray** class (discussed in Chapter 9).

Properties are specified by declaring the type and name of the Property, followed by a bracketed code block that defines a **get** code block (for retrieving the value) and a **set** code block. Read-only properties define only a **get** code block (it is legal, but not obviously useful, to create a write-only property by defining just set). The **get** code block acts as if it were a method defined as taking no arguments and returning the type defined in the Property declaration; the **set** code block acts as if it were a method returning void that takes an argument named **value** of the specified type. Here's an example of a read-write property called **PropertyName** of type **MyType**.

```
class MyClass {
   MyType myInternalReference;

   //Begin property definition
   MyType PropertyName{
      get {
         //logic
         return myInternalReference;
      }

      set{
         //logic
         myInternalReference = value;
      }
   }
   //End of property definition
}
```

To use a Property, you access the name of the property directly:

```
myClassInstance.MyProperty = someValue; //Calls "set"
```

```
MyType t = myClassInstance.MyProperty;  //Calls "get"
```

The C# compiler in fcat *does* create methods in order to implement properties (you'll never need to use them, but if you're interested, the methods are called **get_PropertyName( )** and **set_PropertyName( )**). This is a theme of C# -- direct language support for features that are implemented, not directly in Microsoft Intermediate Language (MSIL – the "machine code" of the .NET runtime), but via code generation. Such "syntactical sugar" could be removed from the C# language without actually changing the set of problems that can be solved by the language; they "just" make tasks easier. Not every language is designed with ease-of-use as a major design goal and some language designers feel that syntactic sugar ends up confusing programmers. For a major language intended to be used by the broadest possible audience, C#'s language design is appropriate; if you want something boiled down to pure functionality, there's talk of LISP being ported to .NET.

Creating New Value Types

In addition to creating new classes, you can create new value types. When you define a value type, the compiler automatically creates the additional data necessary to box and unbox them, so your new value types have all the advantages of both value types and objects.

Enumerations

An enumeration is a set of related values: Up-Down, North-South-East-West, Penny-Nickel-Dime-Quarter, etc. An enumeration is defined using the **enum** keyword and a code block in which the various values are defined. Here's a simple example:

```
enum UpOrDown{ Up, Down }
```

Once defined, an enumeration value can be used by specifying the enumeration type, a dot, and then the specific name desired:

```
UpOrDown coinFlip = UpOrDown.Up;
```

The names within an enumeration are actually numeric values. By default, they are integers, whose value begins at 0. You can modify both the type of storage used for these values and the values associated with a particular

name. Here's an example, where a **short** is used to hold different coin values:

```
enum Coin: short{
   Penny = 1, Nickel = 5, Dime = 10, Quarter = 25
}
```

Then, the names can be cast to their implementing value type:

```
short change = (short) (Coin.Penny + Coin.Quarter);
```

This will result in the value of **change** being 26.

It is also possible to do bitwise operations on enumerations marked with the **[Flags]** attribute:

```
[Flags] enum Flavor{
        Vanilla = 1, Chocolate = 2,
        Strawberry = 4, Coffee = 8}
...etc...
Flavor conePref = Flavor.Vanilla | Flavor.Coffee;
```

Although for bitwise operations, it is of course necessary for you to define the flags with compatible values.

Structs

A **struct** (short for "structure") is very similar to a class in that it can contain fields, properties, and methods. However, **struct**s are value types, you cannot inherit from a **struct** or have your **struct** inherit from any class except **object** (although a **struct** can implement an interface), and **struct**s have limited constructor and destructor semantics.

Typically, **struct**s are used to aggregate a relatively small amount of logically related fields. For instance, the Framework SDK contains a **Point** structure that has **X** and **Y** properties. Structures are declared in the same way as classes. This example shows what might be the start of a **struct** for imaginary numbers:

```
struct ImaginaryNumber{
   double real;
   double Real{
        get { return real; }
        set { real = value; }
```

```
    }
    double i;
    double I{
         get { return i; }
         set { i = value; }
    }

}
```

Building a C# program

There are several other issues you must understand before seeing your first C# program.

Name visibility

A problem in any programming language is the control of names. If you use a name in one module of the program, and another programmer uses the same name in another module, how do you distinguish one name from another and prevent the two names from "clashing?" In C this is a particular problem because a program is often an unmanageable sea of names. C++ classes (on which C# classes are based) nest functions within classes so they cannot clash with function names nested within other classes. However, C++ still allowed global data and global functions, so clashing was still possible. To solve this problem, C++ introduced *namespaces* using additional keywords.

In C#, the **namespace** keyword is followed by a code block (that is, a pair of curly brackets with some amount of code within them). Unlike Java, there is no relationship between the namespace and class names and directory and file structure. Organizationally, it often makes sense to gather all the files associated with a single namespace into a single directory and to have a one-to-one relationship between class names and files, but this is strictly a matter of preference. Throughout this book, our example code will often combine multiple classes in a single compilation unit (that is, a single file) and we will typically not use namespaces, but in professional development, you should avoid such space-saving choices.

Namespaces can, and should, be nested. By convention, the outermost namespace is the name of your organization, the next the name of the project or system as a whole, and the innermost the name of the specific grouping of interest. Here's an example:

```
namespace ThinkingIn{
   namespace CSharp{
      namespace Chap3{
         //class and other type declarations go here
      }
   }
}
```

Since namespaces are publicly viewable, they should start with a capital letter and then use "camelback" capitalization (for instance, **ThinkingIn**).

Namespaces are navigated using dot syntax: **ThinkingIn.CSharp.Chap3** and may even be declared in this manner:

```
namespace ThinkingIn.CSharp.Chap4{ … }
```

Using other components

Whenever you want to use a predefined class in your program, the compiler must know how to locate it. Of course, the class might already exist in the same source code file that it's being called from. In that case, you simply use the class—even if the class doesn't get defined until later in the file. C# eliminates the "forward referencing" problem so you don't need to think about it.

What about a class that exists in some other file? You might think that the compiler should be smart enough to simply go and find it, but there is a problem. Imagine that you want to use a class of a particular name, but more than one definition for that class exists (presumably these are different definitions). Or worse, imagine that you're writing a program, and as you're building it you add a new class to your library that conflicts with the name of an existing class.

To solve this problem, you must eliminate all potential ambiguities. This is accomplished by telling the C# compiler exactly what classes you want

using the **using** keyword. **using** tells the compiler to bring in a *package*, which is a library of classes. (In other languages, a library could consist of functions and data as well as classes, but remember that all code in C# must be written inside a class.)

Most of the time you'll be using components from the .NET Framework SDK. Generally, if one class in a namespace is useful, you'll want to use more than one class from that namespace. This is easily accomplished by the **using** keyword:

```
using System.Collections;
```

It is more common to import a collection of classes in this manner than to import classes individually.

The **static** keyword

Ordinarily, when you create a class you are describing how objects of that class look and how they will behave. You don't actually get anything until you create an object of that class with **new**, and at that point data storage is created and methods become available.

But there are two situations in which this approach is not sufficient. One is if you want to have only one piece of storage for a particular piece of data, regardless of how many objects are created, or even if no objects are created. The other is if you need a method that isn't associated with any particular object of this class. That is, you need a method that you can call even if no objects are created. You can achieve both of these effects with the **static** keyword. When you say something is **static**, it means that data or method is not tied to any particular object instance of that class. So even if you've never created an object of that class you can call a **static** method or access a piece of **static** data. With ordinary, non-**static** data and methods you must create an object and use that object to access the data or method, since non-**static** data and methods must know the particular object they are working with. Of course, since **static** methods don't need any objects to be created before they are used, they cannot *directly* access non-**static** members or methods by simply calling those other members without referring to a named object (since non-**static** members and methods must be tied to a particular object).

Some object-oriented languages use the terms *class data* and *class methods*, meaning that the data and methods exist only for the class as a whole, and not for any particular objects of the class. Sometimes the C# literature uses these terms too.

To make a data member or method **static**, you simply place the keyword before the definition. For example, the following produces a **static** data member and initializes it:

```
class StaticTest {
    static int i = 47;
}
```

Now even if you make two **StaticTest** objects, there will still be only one piece of storage for **StaticTest.i.** Both objects will share the same **i**. Consider:

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

At this point, both **st1.i** and **st2.i** have the same value of 47 since they refer to the same piece of memory.

There are two ways to refer to a **static** variable. As indicated above, you can name it via an object, by saying, for example, **st2.i**. You can also refer to it directly through its class name, something you cannot do with a non-static member. (This is the preferred way to refer to a **static** variable since it emphasizes that variable's **static** nature.)

```
StaticTest.i++;
```

The ++ operator increments the variable. At this point, both **st1.i** and **st2.i** will have the value 48.

Similar logic applies to static methods. You can refer to a static method either through an object as you can with any method, or with the special additional syntax **ClassName.Method( )**. You define a static method in a similar way:

```
class StaticFun {
  static void Incr() { StaticTest.i++; }
}
```

You can see that the **StaticFun** method **Incr( )** increments the **static** data **i**. You can call **Incr( )** in the typical way, through an object:

```
StaticFun sf = new StaticFun();
sf.Incr();
```

Or, because **Incr( )** is a static method, you can call it directly through its class:

```
StaticFun.Incr();
```

While **static**, when applied to a data member, definitely changes the way the data is created (one for each class vs. the non-**static** one for each object), when applied to a method it's not so dramatic. An important use of **static** for methods is to allow you to call that method without creating an object. This is essential, as we will see, in defining the **Main( )** method that is the entry point for running an application.

Like any method, a **static** method can create or use named objects of its type, so a **static** method is often used as a "shepherd" for a flock of instances of its own type.

Putting It All Together

Let's write a program. It starts by printing a string, and then the date, using the **DateTime** class from the .NET Framework SDK. Note that an additional style of comment is introduced here: the '//', which is a comment until the end of the line:

```
// :c03:HelloDate.cs
using System;

namespace ThinkingIn.CSharp.Chap03{
  public class HelloDate {
    public static void Main() {
      Console.WriteLine("Hello, it's: ");
      Console.WriteLine(DateTime.Now);
    }
  }
}///:~
```

At the beginning of each program file, you must place the **using** statement to bring in any classes you'll need for the code in that file.

If you are working with the downloaded .NET Framework SDK, there is a Microsoft Help file that can be accessed with **ms-help://ms.netframeworksdk**, if using Visual Studio .NET, there is an integrated help system. If you navigate to **ms-help://MS.NETFrameworkSDK/cpref/html/frlrfSystem.htm**, you'll see the contents of the **System** namespace. One of them is the **Console** class. If you open this subject and then click on **Console.Members**, you'll see a list of public properties and methods. In the case of the **Console** class, all of them are marked with an "S" indicating that they are static.

One of the static methods of **Console** is **WriteLine( )**. Since it's a static method, you don't need to create an object to use it. Thus, if you've specified **using System;** you can write **Console.WriteLine("Something")** wheneve you want to print something to the console. Alternately, in *any* C# program, you can specify the *fully qualified* name **System.Console.WriteLine("Something")** even if you have not written **using System**.

Every program must have what's called an *entry point*, a method which starts up things. In C#, the entry point is always a static method called **Main( )**. **Main( )** can be written in several different ways:

```
static void Main(){ … }
static void Main(string[] args){ … }
static int Main(){ … }
static int Main(string[] args){ … }
```

If you wish to pass in parameters from the command-line to your program, you should use one of the forms that take an array of command-line arguments. **args[0]** will be the first argument *after* the name of the executable.

Traditionally, programs return a 0 if they ran successfully and some other integer as an error code if they failed. C#'s exceptions are infinitely superior for communicating such problems, but if you are writing a program that you wish to program with batch files, you may wish to use a **Main( )** method that returns an integer.

The line that prints the date illustrates the behind-the-scenes complexity of even a simple object-oriented call:

```
Console.WriteLine(DateTime.Now);
```

Consider the argument: if you browse the documentation to the DateTime structure, you'll discover that it has a static property **Now** of type **DateTime**. As this property is read, the .NET Runtime reads the system clock, creates a new **DateTime** value to store the time, and returns it. As soon as that property **get** finishes, the **DateTime** struct is passed to the static method **WriteLine( )** of the **Console** class. If you use the helpfile to go to that method's definition, you'll see many different *overloaded* versions of **WriteLine( )**, one which takes a **bool**, one which takes a **char**, etc. You won't find one that takes a **DateTime**, though.

Since there is no overloaded version that takes the exact type of the **DateTime** argument, the runtime looks for *ancestors* of the argument type. All **struct**s are defined as descending from type **ValueType**, which in turn descends from type **object**. There is not a version of **WriteLine( )** that takes a **ValueType** for an argument, but there *is* one that takes an **object**. It is this method that is called, passing in the **DateTime** structure.

Back in the documentation for **WriteLine( )**, it says it calls the **ToString( )** method of the **object** passed in as its argument. If you browse to **Object.ToString( )**, though, you'll see that the default representation is just the fully qualified name of the object. But when run, this program doesn't print out "System.DateTime," it prints out the time value itself. This is because the implementers of the **DateTime** class *overrode* the default implementation of **ToString( )** and the call within **WriteLine( )** is resolved *polymorphically* by the **DateTime** implementation, which returns a culture-specific **string** representation of its value to be printed to the **Console**.

If some of that doesn't make sense, don't worry – almost every aspect of object-orientation is at work within this seemingly trivial example.

Compiling and running

To compile and run this program, and all the other programs in this book, you must first have a command-line C# compiler. We *strongly* urge you to refrain from using Microsoft Visual Studio .NET's GUI-activated compiler for compiling the sample programs in this book. The less that is between raw text code and the running program, the more clear the learning experience. Visual Studio .NET introduces additional files to structure and manage projects, but these are not necessary for the small sample programs used in this book. Visual Studio .NET has some great tools that ease certain tasks, like connecting to databases and developing Windows Forms, but these tools should be used to relieve drudgery, not as a substitute for knowledge.

A command-line C# compiler is included in Microsoft's .NET Framework SDK, which is available for free download at msdn.microsoft.com/downloads/ A command-line compiler is also included within Microsoft Visual Studio .NET. The command-line compiler is **csc.exe**. Once you've installed the SDK, you should be able to run **csc** from a command-line prompt.

In addition to the command-line compiler, you should have a decent text editor. Some people seem satisfied with Windows Notepad, but most programmers prefer either the text editor within Visual Studio.NET (just use File/Open... and Save... to work directly with text files) or a third-party programmer's editor. All the code samples in this book were written with Visual SlickEdit from MicroEdge (another favorite is Computer Solution Inc.'s $35 shareware UltraEdit).

Once the Framework SDK is installed, download and unpack the source code for this book (you can find it at www.ThinkingIn.net). This will create a subdirectory for each chapter in the book. Move to the subdirectory **c03** and type:

```
csc HelloDate.cs
```

You should see a message that specifies the versions of the C# compiler and .NET Framework that are being used (the book was finished with C# Compiler version 7.00.9466 and .NET Framework version 1.0.3705). There should be no warnings or errors; if there are, it's an indication that

something went wrong with the SDK installation and you need to investigate those problems.

On the other hand, if you just get your command prompt back, you can type:

```
HelloDate
```

and you'll get the message and the date as output.

This is the process you can use to compile and run each of the programs in this book. A source file, sometimes called a *compilation unit*, is compiled by **csc** into a .NET *assembly*. By default, the assembly is given an extension of **.exe** and, if it has a **Main( )**, the resulting assembly can be run from the command-line just as any other program.

Fine-tuning Compilation

An assembly may be generated from more than one compilation unit. This is done by simply putting the names of the additional compilation units on the command-line (**csc FirstClass.cs SecondClass.cs** etc.). You can modify the name of the assembly with the **/out:** argument. If more than one class has a **Main( )** defined, you can specify which one is intended to be the entry point of the assembly with the **/main:** argument.

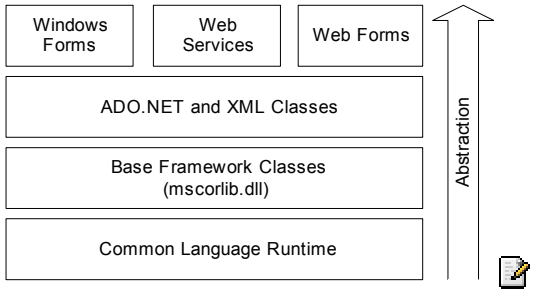
Not every assembly needs to be a stand-alone executable. Such assemblies should be given the **/target:library** argument and will be compiled into an assembly with a **.DLL** extension.

By default, assemblies "know of" the standard library reference **mscorlib.dll**, which contains the majority of the .NET Framework SDK classes. If a program uses a class in a namespace *not* within the **mscorlib.dll** assembly, an assembly containing that class must be either within the current directory, or the **/lib:** argument should be used to point to a directory that holds the appropriate assembly.

The Common Language Runtime

You do not need to know this. But we bet you're curious.

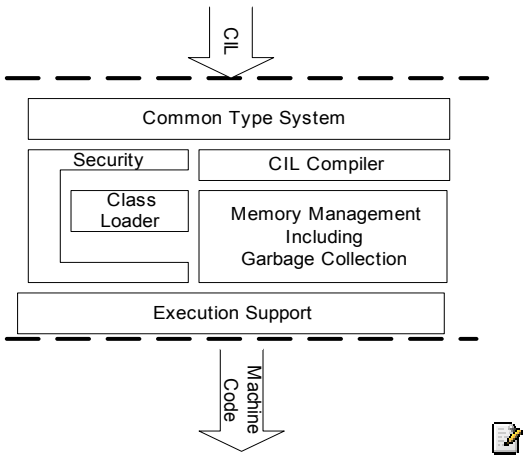
The .NET Framework has several layers of abstraction, from very high-level libraries such as Windows Forms and the SOAP Web Services support, to the core libraries of the SDK:



Everything in this diagram except the Common Language Runtime (CLR) is stored on the computer in Common Intermediate Language (CIL, sometimes referred to as Microsoft Intermediate Language, or MSIL, or sometimes just as IL), a very simple "machine code" for an abstract computer.

The C# compiler, like all .NET language compilers, transforms human-readable source code into CIL, not the actual opcodes of any particular CPU. An assembly consists of CIL, metadata describing the assembly, and optional resources. We'll discuss metadata in detail in Chapter #ref# while resources will be discussed in Chapter 13.

The role of the Common Language Runtime can be boiled down to "mediate between the world of CIL and the world of the actual platform." This requires several components:



Different CPUs and languages have traditionally represented strings in different ways and numeric types using values of different bit-lengths. The value proposition of .NET is "Any language, one platform" (in contrast with Java's value proposition of "Any platform, one language.") In order to assure that all languages can interoperate seamlessly .NET provides a uniform definition of several basic types in the Common Type System. Once "below" this level, the human-readable language in which a module was original written is irrelevant.

Table # shows the transformation of a simple method from C# to CIL to Pentium machine code.

```
class Simple{
   public static void Main(){
      int sum = 0;
      for(int i = 0; i < 5; i++){
         sum += i;
      }
      System.Console.WriteLine(sum);
   }
}
.method public hidebysig static void  Main() cil
managed
{
  .entrypoint
  // Code size       25 (0x19)
  .maxstack  2
```

```
  .locals init (int32 V_0,
           int32 V_1)
  IL_0000:  ldc.i4.0
  IL_0001:  stloc.0
  IL_0002:  ldc.i4.0
  IL_0003:  stloc.1
  IL_0004:  br.s       IL_000e
  IL_0006:  ldloc.0
  IL_0007:  ldloc.1
  IL_0008:  add
  IL_0009:  stloc.0
  IL_000a:  ldloc.1
  IL_000b:  ldc.i4.1
  IL_000c:  add
  IL_000d:  stloc.1
  IL_000e:  ldloc.1
  IL_000f:  ldc.i4.5
  IL_0010:  blt.s      IL_0006
  IL_0012:  ldloc.0
  IL_0013:  call       void
[mscorlib]System.Console::WriteLine(int32)
  IL_0018:  ret
} // end of method Simple::Main
```

Security restrictions are implemented at this level in order to make it extremely difficult to bypass. To seamlessly bypass security would require replacing the CLR with a hacked CLR, not impossible to conceive, but hopefully beyond the range of script kiddies and requiring an administration-level compromise from which to start. The security model of .NET consists of checks that occur at both the moment the class is loaded into memory and at the moment that possibly-restricted operations are requested.

Although CIL is not representative of any real machine code, it is *not* interpreted. After the CIL of a class is loaded into memory, a Just-In-Time compiler (JIT) transforms it from CIL into machine language appropriate to the native CPU. One interesting benefit of this is that it's conceivable that different JIT compilers might become available for different CPUs within a general family (thus, we might eventually have an Itanium JIT, a Pentium JIT, an Athlon JIT, etc.).

The CLR contains a subsystem responsible for memory management inside what is called "managed code." In addition to garbage collection (the process of recycling memory), the CLR memory manager defragments memory and decreases the span of reference of in-memory references (both of which are beneficial side effects of the garbage collection architecture).

Finally, all programs require some basic execution support at the level of thread scheduling, code execution, and other system services. Once again, at this low level, all of this support can be shared by any .NET application, no matter what the originating programming language.

The Common Language Runtime, the base framework classes within mscorlib.dll, and the C# language have all been submitted to Microsoft to the European Computer Manufacturers Association (ECMA) for ratification as standards. The Mono Project (www.go-mono.com) is an effort to create an Open Source implementation of these standards that includes Linux support.

Comments and embedded documentation

There are two types of comments in C#. The first is the traditional C-style comment that was inherited by C++. These comments begin with a /* and continue, possibly across many lines, until a */. Note that many programmers will begin each line of a continued comment with a *, so you'll often see:

```
/* This is a comment
*  that continues
*  across lines
*/
```

Remember, however, that everything inside the /* and */ is ignored, so there's no difference in saying:

```
/* This is a comment that
continues across lines */
```

The second form of comment comes from C++. It is the single-line comment, which starts at a // and continues until the end of the line. This type of comment is convenient and commonly used because it's easy. You don't need to hunt on the keyboard to find / and then * (instead, you just press the same key twice), and you don't need to close the comment. So you will often see:

```
// this is a one-line comment
```

Documentation Comments

One of the thoughtful parts of the C# language is that the designers didn't consider writing code to be the only important activity—they also thought about documenting it. Possibly the biggest problem with documenting code has been maintaining that documentation. If the documentation and the code are separate, it becomes a hassle to change the documentation every time you change the code. The solution seems simple: link the code to the documentation. The easiest way to do this is to put everything in the same file. To complete the picture, however, you need a special comment syntax to mark special documentation, and a tool to extract those comments and put them in a useful form. This is what C# has done.

Comments that begin with /// can be extracted from source files by running **csc /doc:OutputFile.XML**. Inside the comments you can place any valid XML tags including some tags with predefined meanings discussed next. The resulting XML file is interpreted in certain ways inside of Visual Studio .NET or can be styled with XSLT to produce a Web page or printable documentation. If you don't understand XML, don't worry about it; you'll become much more familiar with it in Chapter 14!

If you run

```
csc /doc:HelloDate.xml HelloDate.cs
```

The resulting XML will be:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>HelloDate</name>
```

```
    </assembly>
    <members>
    </members>
</doc>
```

The XML consists of a "doc" element, which is for the assembly named "HelloDate" and which doesn't have any documentation comments.

Tag	Suggested Use
<summary> </summary>	A brief overview of the code element
<remarks> </remarks>	This is used for a more comprehensive discussion of the element's intended behavior
<param name="name"> </param>	One of these tags should be written for each argument to a method; the value of the **name** attribute specifies which argument. The description should include any preconditions associated with the argument. Preconditions are what the method requires of its arguments so that the method can function correctly. For instance, a precondition of a square root function might be that the input integer be positive.
<returns> </returns>	Methods that return anything other than void should have one of these tags. The contents of the tag should describe what about the return value can be guaranteed. Can it be null? Does it always fall within a certain range? Is it always in a certain state? etc.
<exception cref="type"> </exception>	Every exception that is explicitly raised within the method's body should be documented in a tag such as this (the type of the exception should be the value of the **cref** attribute). To the extent possible, the circumstances which give rise to the exception being thrown should be detailed. Because of C#'s exception model (discussed in chapter #ref#), special attention should be paid to making sure that these comments are consistently and uniformly written and maintained.

<permission cref="type"> </permission>	This tag describes the security permissions that are required for the type. The **cref** attribute is optional, but if it exists, it should refer to a PermissionSet associated with the type.
<example> <c></c> <code></code> </example>	The <example> tag should contain a description of a sample use of the code element. The <c> tag is intended to specify an inline code element while the <code> tag is intended for multiline snippets.
<see cref="other"> </see> <seealso cref="other"> </seealso>	These tags are intended for cross references to other code elements or other documentation fragments. The <see> tag is intended for inline cross-references, while the <seealso> tag is intended to be broken out into a separate "See Also" section
<value> </value>	Every externally visible property within a class should be documented with this tag
<paramref name="arg"/>	This empty tag is used when commenting a method to indicate that the value of the **name** attribute is actually the name of one of the method's arguments.
<list type= [bullet | number | table]> <listheader> <term></term> <description> </description> </listheader> <item> <term></term> <description> </description> </item> </list>	Intended to provide a hint to the XML styler on how to generate documentation.

<para></para>	Intended to specify separate paragraphs within a description or other lengthy text block

Documentation example

Here's the HelloDate C# program, this time with documentation comments added:

```
// :c03:HelloDate2.cs
using System;

namespace ThinkingIn.CSharp.Chap03{
   ///<summary>Shows doc comments</summary>
   ///<remarks>The documentation comments within C#
   ///are remarkably useful, both within the Visual
   ///Studio environment and as the basis for more
   ///significant printed documentation</remarks>
  public class HelloDate2 {

    ///<summary>Entry point</summary>
    ///<remarks>Prints greeting to
    /// <paramref name="args[0]"/>, gets a
    /// <see cref="System.DateTime">DateTime</see>
    /// and subsequently prints it</remarks>
    ///<param name="args">Command-line should have a
    ///single name. All other args will be ignored
    ///</param>
    public static void Main(string[] args) {
      Console.WriteLine("Hello, {0} it's: ", args[0]);
      Console.WriteLine(DateTime.Now);
    }
  }
}//:~
```

When **csc** extracts the data, it is in this form:

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>HelloDate</name>
  </assembly>
  <members>
```

```
    <member
      name="T:ThinkingIn.CSharp.Chap03.HelloDate2">
      <summary>Shows doc comments</summary>
      <remarks>The documentation comments within C#
      are remarkably useful, both within the Visual
      Studio environment and as the basis for more
      significant printed documentation</remarks>
    </member>
    <member
name="M:ThinkingIn.CSharp.Chap03.HelloDate2.Main(Syste
m.String[])">
      <summary>Entry point</summary>
      <remarks>Prints greeting to
       <paramref name="args[0]"/>, gets a
       <see cref="T:System.DateTime">DateTime</see>
       and subsequently prints it</remarks>
      <param name="args">Command-line should have a
      single name. All other args will be ignored
      </param>
    </member>
  </members>
</doc>
```

The first line of the HelloDate2.cs file uses a convention that will be used throughout the book. Every compilable sample begins with a comment followed by a ':' the chapter number, another colon, and the name of the file that the example should be saved to. The last line also finishes with a comment, and this one indicates the end of the source code listing, which allows it to be automatically extracted from the text of this book and checked with a compiler. This convention supports a tool which can automatically extract and compile code directly from the "source" Word document.

Coding style

The unofficial standard in C# is to capitalize the first letter of a class name. If the class name consists of several words, they are run together (that is, you don't use underscores to separate the names), and the first letter of each embedded word is capitalized, such as:

```
class AllTheColorsOfTheRainbow { // ...
```

This same style is also used for the parts of the class which are intended to be refered to by others (method names and properties). For internal parts fields (member variables) and object reference names, the accepted style is just as it is for classes *except* that the first letter of the identifier is lowercase. For example:

```
class AllTheColorsOfTheRainbow {
  int anIntegerRepresentingColors;
  void ChangeTheHueOfTheColor(int newHue) {
    // ...
  }
  // ...
}
```

Of course, you should remember that the user must also type all these long names, and so be merciful.

Summary

In this chapter you have seen enough of C# programming to understand how to write a simple program, and you have gotten an overview of the language and some of its basic ideas. However, the examples so far have all been of the form "do this, then do that, then do something else." What if you want the program to make choices, such as "if the result of doing this is red, do that; if not, then do something else"? The support in C# for this fundamental programming activity will be covered in the next chapter.

Exercises

1. Following the **HelloDate.cs** example in this chapter, create a "hello, world" program that simply prints out that statement. You need only a single method in your class (the "Main" one that gets executed when the program starts). Remember to make it **static**. Compile the program with **csc** and run it from the command-line.

2. Find the code fragments involving **ATypeName** and turn them into a program that compiles and runs.

3. Turn the **DataOnly** code fragments into a program that compiles and runs.
4. Modify Exercise 3 so that the values of the data in **DataOnly** are assigned to and printed in **Main( )**.
5. Write a program that includes and calls the **Storage( )** method defined as a code fragment in this chapter.
6. Turn the **StaticFun** code fragments into a working program.
7. Write a program that prints three arguments taken from the command line. To do this, you'll need to index into the command-line array of **string**s.
8. Turn the **AllTheColorsOfTheRainbow** example into a program that compiles and runs.
9. Find the code for the second version of **HelloDate.cs**, which is the simple comment documentation example. Execute **csc /doc** on the file and view the results with your XML-aware Web browser.
10. Turn **DocTest** into a file that compiles and then compile it with **csc /doc**. Verify the resulting documentation with your Web browser.
11. Add an HTML list of items to the documentation in Exercise 10.
12. Take the program in Exercise 1 and add comment documentation to it. Extract this comment documentation and view it with your Web browser.

4: Controlling Program Flow

@todo: Break first part into a chapter called, maybe "Types and operators" and the second part (starting at "Flow Control") into "Controlling program flow."

@todo: Short section on reference vs. value types. Maybe some diagrams on stack, heap, pointers, etc.?

Like a sentient creature, a program must manipulate its world and make choices during execution.

In C# you manipulate objects and data using operators, and you make choices with execution control statements. The statements used will be familiar to programmers with Java, C++, or C backgrounds, but there are a few that may seem unusual to programmers coming from Visual Basic backgrounds.

Using C#operators

An operator takes one or more arguments and produces a new value. The arguments are in a different form than ordinary method calls, but the effect is the same. You should be reasonably comfortable with the general concept of operators from your previous programming experience. Addition (+), subtraction and unary minus (-), multiplication (*), division (/), and assignment (=) all work much the same in any programming language.

All operators produce a value from their operands. In addition, an operator can change the value of an operand. This is called a *side effect*. The most common use for operators that modify their operands is to generate the side effect, but you should keep in mind that the value

produced is available for your use just as in operators without side effects.

Operators work with all primitives and many objects. When you program your own objects, you will be able to extend them to support whichever primitives make sense (you'll find yourself creating '+' operations far more often than '/' operations!) The operators '**=**', '**==**' and '**!=**', work for all objects and are a point of confusion for objects that we'll deal with in #reference#.

Precedence

Operator precedence defines how an expression evaluates when several operators are present. C# has specific rules that determine the order of evaluation. The easiest one to remember is that multiplication and division happen before addition and subtraction. Programmers often forget the other precedence rules, so you should use parentheses to make the order of evaluation explicit. For example:

```
a = x + y - 2/2 + z;
```

has a very different meaning from the same statement with a particular grouping of parentheses:

```
a = x + (y - 2)/(2 + z);
```

Assignment

Assignment is performed with the operator =. It means "take the value of the right-hand side (often called the *rvalue*) and copy it into the left-hand side (often called the *lvalue*). An rvalue is any constant, variable or expression that can produce a value, but an lvalue must be a distinct, named variable. (That is, there must be a physical space to store a value.) For instance, you can assign a constant value to a variable (**A = 4;**), but you cannot assign anything to constant value—it cannot be an lvalue. (You can't say **4 = A;**.)

Assignment of primitives is quite straightforward. Since the primitive holds the actual value and not a reference to an object, when you assign primitives you copy the contents from one place to another. For example, if you say **A = B** for primitives, then the contents of **B** are copied into **A**. If

you then go on to modify **A**, **B** is naturally unaffected by this modification. As a programmer, this is what you've come to expect for most situations.

When you assign objects, however, things change. Whenever you manipulate an object, what you're manipulating is the reference, so when you assign "from one object to another" you're actually copying a reference from one place to another. This means that if you say **C = D** for objects, you end up with both **C** and **D** pointing to the object that, originally, only **D** pointed to. The following example will demonstrate this.

Here's the example:

```
//: c03: Assignment.cs
class Number{
    public int i;
}

public class Assignment{
    public static void Main(){
        Number n1 = new Number();
        Number n2 = new Number();
        n1.i = 9;
        n2.i = 47;
        System.Console.WriteLine(
            "1: n1.i: " + n1.i + ", n2.i: " + n2.i);
        n1.i = n2.i;
        System.Console.WriteLine(
            "2: n1.i: " + n1.i + ", n2.i: " + n2.i);
        n1.i = 27;
        System.Console.WriteLine(
            "3: n1.i: " + n1.i + ", n2.i: " + n2.i);
    }
}
///:~
```

The **Number** class is simple, and two instances of it (**n1** and **n2**) are created within M**ain( )**. The **i** value within each **Number** is given a different value, and then **n2** is assigned to **n1**, and **n1** is changed. In many programming languages you would expect **n1** and **n2** to be independent

at all times, but because you've assigned a reference here's the output you'll see:

```
1: n1.i: 9, n2.i: 47
2: n1.i: 47, n2.i: 47
3: n1.i: 27, n2.i: 27
```

Changing the **n1** object appears to change the **n2** object as well! This is because both **n1** and **n2** contain the same reference, which is pointing to the same object. (The original reference that was in **n1** that pointed to the object holding a value of 9 was overwritten during the assignment and effectively lost; its object will be cleaned up by the garbage collector.)

This phenomenon is called *aliasing* and it's a fundamental way that C# works with objects. But what if you don't want aliasing to occur in this case? You could forego the assignment and say:

```
n1.i = n2.i;
```

This retains the two separate objects instead of tossing one and tying **n1** and **n2** to the same object, but you'll soon realize that manipulating the fields within objects is messy and goes against good object-oriented design principles.

Aliasing during method calls

Aliasing will also occur when you pass an object into a method:

```
//: c03: PassObject.cs
namespace c03{
    class Letter{
        public char c;
    }

    public class PassObject{
        static void f(Letter y){
            y.c = 'z';
        }

        public static void Main(){
            Letter x = new Letter();
            x.c = 'a';
```

```
            System.Console.WriteLine("1: x.c: " +
x.c);
            f(x);
            System.Console.WriteLine("2: x.c: " +
x.c);
        }
    }
}///:~
```

In many programming languages, the method **F( )** would appear to be making a copy of its argument **Letter y** inside the scope of the method. But once again a reference is being passed so the line

```
y.c = 'z';
```

is actually changing the object outside of **F( )**. The output shows this:

```
1: x.c: a
2: x.c: z
```

Mathematical operators

The basic mathematical operators are the same as the ones available in most programming languages: addition (+), subtraction (-), division (/), multiplication (*) and modulus (**%**, which produces the remainder from integer division). Integer division truncates, rather than rounds, the result.

Java also uses a shorthand notation to perform an operation and an assignment at the same time. This is denoted by an operator followed by an equal sign, and is consistent with all the operators in the language (whenever it makes sense). For example, to add 4 to the variable **x** and assign the result to **x**, use: **x += 4**.

This example shows the use of the mathematical operators:

```
//: c03:MathOps.cs
namespace c03{
    using System;

    public class MathOps{
```

```
        ///Create a shorthand function to save typing
        static void Prt(String s){
            System.Console.WriteLine(s);
        }

        ///Prints a string and an int:
        static void PInt(String s, int i){
            Prt(s + " = " + i);
        }

        //Shorthand to print a string and a float
        static void PDouble(String s, double f){
            Prt(s + " = " + f);
        }

        public static void Main(){
            //Create a random number generator,
            //seeds with current time by default
            Random rand = new Random();
            int i, j, k;
            //get a positive random number less than
the specified maximum
            j = rand.Next(100);
            k = rand.Next(100);
            PInt("j",j); PInt("k",k);
            i = j + k; PInt("j + k", i);
            i = j - k; PInt("j - k", i);
            i = k / j; PInt("k / j", i);
            i = k * j; PInt("k * j", i);
            //Limits i to a positive number less than
j
            i = k % j; PInt("k % j", i);
            j %= k; PInt("j %= k", j);
            //Floating-point number tests:
            double u, v, w;
            v = rand.NextDouble();
            w= rand.NextDouble();
            PDouble("v", v); PDouble("w",w);
            u = v + w; PDouble("v + w", u);
            u = v - w; PDouble("v - w", u);
            u = v * w; PDouble("v * w", u);
```

```
            u = v / w; PDouble("v / w", u);
            //the following also works for
            //char, byte, short, int, long,
            //and float @todo check
            u += v; PDouble("u += v", u);
            u -= v; PDouble("u -= v", u);
            u *= v; PDouble("u *= v", u);
            u /= v; PDouble("u /= v", u);
        }
    }
}///:~
```

The first thing you will see are some shorthand methods for printing: the **Prt( )** method prints a **String**, the **PInt( )** prints a **String** followed by an **int** and the **PDouble( )** prints a **String** followed by a **double**. Of course, they all ultimately end up using **System.Console.WriteLine( )**.

To generate numbers, the program first creates a **Random** object. Because no arguments are passed during creation, C# uses the current time as a seed for the random number generator. The program generates a number of different types of random numbers with the **Random** object simply by calling the methods: **Next ( )** and **NextDouble( )** (you can also call N**extLong( )** or **Next(int)**).

Unary minus and plus operators

The unary minus (-) and unary plus (+) are the same operators as binary minus and plus. The compiler figures out which use is intended by the way you write the expression. For instance, the statement

```
x = -a;
```

has an obvious meaning. The compiler is able to figure out:

```
x = a * -b;
```

but the reader might get confused, so it is clearer to say:

```
x = a * (-b);
```

The unary minus produces the negative of the value. Unary plus provides symmetry with unary minus, although it doesn't have any effect.

Auto increment and decrement

C#, like C, is full of shortcuts. Shortcuts can make code much easier to type, and either easier or harder to read.

Two of the nicer shortcuts are the increment and decrement operators (often referred to as the auto-increment and auto-decrement operators). The decrement operator is **--** and means "decrease by one unit." The increment operator is **++** and means "increase by one unit." If **a** is an **int**, for example, the expression **++a** is equivalent to (**a = a + 1**). Increment and decrement operators produce the value of the variable as a result.

There are two versions of each type of operator, often called the prefix and postfix versions. Pre-increment means the **++** operator appears before the variable or expression, and post-increment means the **++** operator appears after the variable or expression. Similarly, pre-decrement means the **--** operator appears before the variable or expression, and post-decrement means the **--** operator appears after the variable or expression. For pre-increment and pre-decrement, (i.e., **++a** or **--a**), the operation is performed and the value is produced. For post-increment and post-decrement (i.e. **a++** or **a--**), the value is produced, then the operation is performed. As an example:

```
//: c03:AutoInc.cs
namespace c03{
    using System;

    public class AutoInc{
        public static void Main(){
            int i = 1;
            prt("i: " + i);
            prt("++i: " + ++i); //Pre-increment
            prt("i++: " + i++); //Post-increment
            prt("i: " + i);
            prt("--i: " + --i); //Pre-increment
```

```
            prt("i--: " + i--); //Post-increment
            prt("i: " + i);
        }

        static void prt(String s){
            System.Console.WriteLine(s);
        }
    }
}///:~
```

The output for this program is:

```
i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1
```

You can see that for the prefix form you get the value after the operation has been performed, but with the postfix form you get the value before the operation is performed. These are the only operators (other than those involving assignment) that have side effects. (That is, they change the operand rather than using just its value.)

The increment operator is one explanation for the name C++, implying "one step beyond C." As for C#, the explanation seems to be in music, where the # symbolizes "sharp" – a half-step "up."

Relational operators

Relational operators generate a boolean result. They evaluate the relationship between the values of the operands. A relational expression produces **true** if the relationship is true, and **false** if the relationship is untrue. The relational operators are less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=), equivalent (==) and not equivalent (!=). Equivalence and nonequivalence work with all built-in data types, but the other comparisons won't work with type **bool**.

Testing object equivalence

The relational operators == and **!=** also work with all objects, but their meaning often confuses the first-time C# programmer. Here's an example:

```
//: c03:EqualsOperator.cs
 namespace c03{
    using System;
    class MyInt{
        Int32 i;
        public MyInt(int j){
            i = j;
        }
    }
    ///Demonstrates handle inequivalence.
    public class EqualsOperator{
        public static void Main(){
            MyInt m1 = new MyInt(47);
            MyInt m2 = new MyInt(47);
            System.Console.WriteLine("m1 == m2: " +
(m1 == m2));
        }
    }
}///:~
```

The expression **System.Console.WriteLine(n1 == n2)** will print the result of the **bool** comparison within it. Surely the output should be **true** and then **false**, since both **MyInt** objects have the same value. But while the *contents* of the objects are the same, the references are not the same and the operators == and **!=** compare object references. So the output is actually **false** and then **true**. Naturally, this surprises people at first.

What if you want to compare the actual contents of an object for equivalence? For objects in a well-designed class library (such as the .NET framework), you just use the equivalence operators. However, if you create your own class, you must *override* '==' in your new class to get the desired behavior. Unfortunately, you won't learn about overriding until Chapter 7, but being aware of the way '==' behaves might save you some grief in the meantime.

Logical operators

Each of the logical operators AND (&&), OR (||) and NOT (!) produces a **bool** value of **true** or **false** based on the logical relationship of its arguments. This example uses the relational and logical operators:

```
//: c03:Bool.cs
namespace c03{
    using System;
// Relational and logical operators.
public class Bool {
  public static void Main() {
    Random rand = new Random();
    int i = rand.Next(100);
    int j = rand.Next(100);
    prt("i = " + i);
    prt("j = " + j);
    prt("i > j is " + (i > j));
    prt("i < j is " + (i < j));
    prt("i >= j is " + (i >= j));
    prt("i <= j is " + (i <= j));
    prt("i == j is " + (i == j));
    prt("i != j is " + (i != j));

    // Treating an int as a boolean is
    // not legal C#
//!    prt("i && j is " + (i && j));
//!    prt("i || j is " + (i || j));
//!    prt("!i is " + !i);

    prt("(i < 10) && (j < 10) is "
       + ((i < 10) && (j < 10)) );
    prt("(i < 10) || (j < 10) is "
       + ((i < 10) || (j < 10)) );
  }
  static void prt(String s) {
    System.Console.WriteLine(s);
  }
}
}///:~
```

You can apply AND, OR, or NOT to **bool** values only. You can't use a non-**bool** as if it were a **bool** in a logical expression as you can in some other languages. You can see the failed attempts at doing this commented out with a **//!** comment marker. The subsequent expressions, however, produce **bool** values using relational comparisons, then use logical operations on the results.

One output listing looked like this:

```
i = 85
j = 4
i > j is true
i < j is false
i >= j is true
i <= j is false
i == j is false
i != j is true
(i < 10) && (j < 10) is false
(i < 10) || (j < 10) is true
```

Note that a **bool** value is automatically converted to an appropriate text form if it's used where a **string** is expected.

You can replace the definition for **int** in the above program with any other primitive data type except **bool**. Be aware, however, that the comparison of floating-point numbers is very strict. A number that is the tiniest fraction different from another number is still "not equal." A number that is the tiniest bit above zero is still nonzero.

Short-circuiting

When dealing with logical operators you run into a phenomenon called "short circuiting." This means that the expression will be evaluated only *until* the truth or falsehood of the entire expression can be unambiguously determined. As a result, all the parts of a logical expression might not be evaluated. Here's an example that demonstrates short-circuiting:

```
//: c03:ShortCircuit.cs
namespace c03{
    using System;
// Demonstrates short-circuiting behavior.
```

```
// with logical operators.
    public class ShortCircuit {
        static bool Test1(int val) {
            System.Console.WriteLine("Test1(" + val +
")");
            System.Console.WriteLine("result: " + (val
< 1));
            return val < 1;
        }
        static bool Test2(int val) {
            System.Console.WriteLine("Test2(" + val +
")");
            System.Console.WriteLine("result: " + (val
< 2));
            return val < 2;
        }
        static bool Test3(int val) {
            System.Console.WriteLine("Test3(" + val +
")");
            System.Console.WriteLine("result: " + (val
< 3));
            return val < 3;
        }
        public static void Main() {
            if (Test1(0) && Test2(2) && Test3(2))
                System.Console.WriteLine("expression
is true");
            else
                System.Console.WriteLine("expression
is false");
        }
    }
} ///:~
```

Each test performs a comparison against the argument and returns true or false. It also prints information to show you that it's being called. The tests are used in the expression:

```
if(test1(0) && test2(2) && test3(2))
```

You might naturally think that all three tests would be executed, but the output shows otherwise:

```
test1(0)
result: true
test2(2)
result: false
expression is false
```

The first test produced a **true** result, so the expression evaluation continues. However, the second test produced a **false** result. Since this means that the whole expression must be **false**, why continue evaluating the rest of the expression? It could be expensive. The reason for short-circuiting, in fact, is precisely that; you can get a potential performance increase if all the parts of a logical expression do not need to be evaluated.

Bitwise operators

The bitwise operators allow you to manipulate individual bits in an integral primitive data type. Bitwise operators perform boolean algebra on the corresponding bits in the two arguments to produce the result.

The bitwise operators come from C's low-level orientation; you were often manipulating hardware directly and had to set the bits in hardware registers. Although most application and Web Service developers will not be using the bitwise operators much, developers for Handheld PCs, set-top boxes, and the X Box often need every bit-twiddling advantage they can get.

The bitwise AND operator (**&**) produces a one in the output bit if both input bits are one; otherwise it produces a zero. The bitwise OR operator (|) produces a one in the output bit if either input bit is a one and produces a zero only if both input bits are zero. The bitwise EXCLUSIVE OR, or XOR (^), produces a one in the output bit if one or the other input bit is a one, but not both. The bitwise NOT (~, also called the *ones complement* operator) is a unary operator; it takes only one argument. (All other bitwise operators are binary operators.) Bitwise NOT produces

the opposite of the input bit—a one if the input bit is zero, a zero if the input bit is one.

The bitwise operators and logical operators use the same characters, so it is helpful to have a mnemonic device to help you remember the meanings: since bits are "small," there is only one character in the bitwise operators.

Bitwise operators can be combined with the = sign to unite the operation and assignment: **&**=, |= and ^= are all legitimate. (Since ~ is a unary operator it cannot be combined with the = sign.)

The **bool** type is treated as a one-bit value so it is somewhat different. You can perform a bitwise AND, OR and XOR, but you can't perform a bitwise NOT (presumably to prevent confusion with the logical NOT). For **bool**s the bitwise operators have the same effect as the logical operators except that they do not short circuit. Also, bitwise operations on **bool**s include an XOR logical operator that is not included under the list of "logical" operators. You're prevented from using **bool**s in shift expressions, which is described next.

Shift operators

The shift operators also manipulate bits. They can be used solely with primitive, integral types. The left-shift operator (<<) produces the operand to the left of the operator shifted to the left by the number of bits specified after the operator (inserting zeroes at the lower-order bits). The signed right-shift operator (>>) produces the operand to the left of the operator shifted to the right by the number of bits specified after the operator. The signed right shift >> uses *sign extension*: if the value is positive, zeroes are inserted at the higher-order bits; if the value is negative, ones are inserted at the higher-order bits. (C# does not have an unsigned right shift >>>.)

If you shift a **char**, **byte,** or **short**, it will be promoted to **int** before the shift takes place, and the result will be an **int**. Only the five low-order bits of the right-hand side will be used. This prevents you from shifting more than the number of bits in an **int**. If you're operating on a **long**, you'll get a **long** result. Only the six low-order bits of the right-hand side will be used so you can't shift more than the number of bits in a **long**.

Shifts can be combined with the equal sign (<<= or >>=). The lvalue is replaced by the lvalue shifted by the rvalue.

Here's an example that demonstrates the use of all the operators involving bits:

```
//: c03:BitManipulation.cs
namespace c03{
    using System;

public class BitManipulation {
  public static void Main() {
    Random rand = new Random();
    int i = rand.Next();
    int j = rand.Next();
    PBinInt("-1", -1);
    PBinInt("+1", +1);
    int maxpos = Int32.MaxValue;
    PBinInt("maxpos", maxpos);
    int maxneg = Int32.MinValue;
    PBinInt("maxneg", maxneg);
    PBinInt("i", i);
    PBinInt("~i", ~i);
    PBinInt("-i", -i);
    PBinInt("j", j);
    PBinInt("i & j", i & j);
    PBinInt("i | j", i | j);
    PBinInt("i ^ j", i ^ j);
    PBinInt("i << 5", i << 5);
    PBinInt("i >> 5", i >> 5);
    PBinInt("(~i) >> 5", (~i) >> 5);
    ///@todo Unsigned shifts don't work!
    /*
    PBinInt("i >>> 5", i >>> 5);
    PBnInt("(~i) >>> 5", (~i) >>> 5);
    */
    long l_high_bits = rand.Next();
    l_high_bits <<= 32;
    long l = l_high_bits + rand.Next();
    long m_high_bits = rand.Next();
    m_high_bits <<=32;
```

```
    long m = m_high_bits + rand.Next();
    PBinLong("-1L", -1L);
    PBinLong("+1L", +1L);
    long ll = Int64.MaxValue;
    PBinLong("maxpos", ll);
    long lln = Int64.MinValue;
    PBinLong("maxneg", lln);
    PBinLong("l_high_bits", l_high_bits);
    PBinLong("l", l);
    PBinLong("~l", ~l);
    PBinLong("-l", -l);
    PBinLong("m_high_bits", m_high_bits);
    PBinLong("m", m);
    PBinLong("l & m", l & m);
    PBinLong("l | m", l | m);
    PBinLong("l ^ m", l ^ m);
    PBinLong("l << 5", l << 5);
    PBinLong("l >> 5", l >> 5);
    PBinLong("(~l) >> 5", (~l) >> 5);
    /* @todo
    PBinLong("l >>> 5", l >>> 5);
    PBinLong("(~l) >>> 5", (~l) >>> 5);
    */
  }
  static void PBinInt(String s, int i) {
    System.Console.WriteLine(
      s + ", int: " + i + ", binary: ");
    System.Console.Write("   ");
    for(int j = 31; j >=0; j--)
      if(((1 << j) &  i) != 0)
        System.Console.Write("1");
      else
        System.Console.Write("0");
    System.Console.WriteLine();
  }

  static void PBinLong(String s, long l) {
    System.Console.WriteLine(
      s + ", long: " + l + ", binary: ");
    System.Console.Write("   ");
    for(int i = 63; i >=0; i--)
```

```
      if(((1L << i) & l) != 0)
        System.Console.Write("1");
      else
        System.Console.Write("0");
    System.Console.WriteLine();
  }
}
}
//:~
```

The two methods at the end, P**BinInt( )** and **PBinLong( )** take an **int** or a **long**, respectively, and print it out in binary format along with a descriptive string. You can ignore the implementation of these for now.

You'll note the use of **System.Console.Write( )** instead of **System.Console.WriteLine( )**. The **Write( )** method does not emit a new line, so it allows you to output a line in pieces.

As well as demonstrating the effect of all the bitwise operators for **int** and **long**, this example also shows the minimum, maximum, +1 and -1 values for **int** and **long** so you can see what they look like. Note that the high bit represents the sign: 0 means positive and 1 means negative. The output looks like this:

```
-1, int: -1, binary:
   11111111111111111111111111111111
+1, int: 1, binary:
   00000000000000000000000000000001
maxpos, int: 2147483647, binary:
   01111111111111111111111111111111
maxneg, int: -2147483648, binary:
   10000000000000000000000000000000
i, int: 1177419330, binary:
   01000110001011011111111001000010
~i, int: -1177419331, binary:
   10111001110100100000000110111101
-i, int: -1177419330, binary:
   10111001110100100000000110111110
j, int: 886693932, binary:
   00110100110110011110000000101100
i & j, int: 67756032, binary:
```

```
   00000100000010011110000000000000
i | j, int: 1996357230, binary:
   01110110111111011111111001101110
i ^ j, int: 1928601198, binary:
   01110010111101000001111001101110
i << 5, int: -977287104, binary:
   11000101101111111100100001000000
i >> 5, int: 36794354, binary:
   00000010001100010110111111110010
(~i) >> 5, int: -36794355, binary:
   11111101110011101001000000001101
-1L, long: -1, binary:

1111111111111111111111111111111111111111111111111111111111
1111111111
+1L, long: 1, binary:

0000000000000000000000000000000000000000000000000000000000
0000000001
maxpos, long: 9223372036854775807, binary:

0111111111111111111111111111111111111111111111111111111111
1111111111
maxneg, long: -9223372036854775808, binary:

1000000000000000000000000000000000000000000000000000000000
0000000000
l_high_bits, long: 4654972597212020736, binary:

0100000010011001110010001110101000000000000000000000000000
0000000000
l, long: 4654972598829014295, binary:

0100000010011001110010001110101001100000011000010101110100
0100010111
~l, long: -4654972598829014296, binary:

1011111101100110001101110001010110011111100111010100010111
1011101000
-l, long: -4654972598829014295, binary:
```

```
1011111101100110001101110001010110011111100111101010001011101001
m_high_bits, long: 468354230734815232, binary:

0000011001111111111011011011000100000000000000000000000000000000
m, long: 468354231158705547, binary:

0000011001111111111011011011000100011001010001000000110110001011
l & m, long: 7257463942286595, binary:

0000000000011001110010001010000000000000010000000000110100000011
l | m, long: 5116069366045433247, binary:

0100011011111111111011011111101101111001011001010101110110011111
l ^ m, long: 5108811902103146652, binary:

0100011011100110001001010101101101111001001001010101001000010011100
l << 5, long: 1385170572852044512, binary:

0001001100111001000111010100110000001100001010111010001011100000
l >> 5, long: 145467893713406696, binary:

0000001000000100110011100100011101010011000000110000101011101000
(~l) >> 5, long: -145467893713406697, binary:

1111110111111011001100011011100010101100111111001111010100010111
```

The binary representation of the numbers is referred to as *signed two's complement*.

Ternary if-else operator

This operator is unusual because it has three operands. It is truly an operator because it produces a value, unlike the ordinary if-else statement that you'll see in the next section of this chapter. The expression is of the form:

```
boolean-exp ? value0 : value1
```

If *boolean-exp* evaluates to **true**, *valueo* is evaluated and its result becomes the value produced by the operator. If *boolean-exp* is **false**, *value1* is evaluated and its result becomes the value produced by the operator.

Of course, you could use an ordinary **if-else** statement (described later), but the ternary operator is much terser. Although C (where this operator originated) prides itself on being a terse language, and the ternary operator might have been introduced partly for efficiency, you should be somewhat wary of using it on an everyday basis—it's easy to produce unreadable code.

The conditional operator can be used for its side effects or for the value it produces, but in general you want the value since that's what makes the operator distinct from the **if-else**. Here's an example:

```
static int Ternary(int i) {
  return i < 10 ? i * 100 : i * 10;
}
```

You can see that this code is more compact than what you'd need to write without the ternary operator:

```
static int Alternative(int i) {
  if (i < 10)
    return i * 100;
  else
    return i * 10;
}
```

The second form is easier to understand, and doesn't require a lot more typing. So be sure to ponder your reasons when choosing the ternary operator – it's generally warranted when you're setting a variable to one of two values:

```
int ternaryResult = i < 10 ? i * 100 : i * 10;
```

The comma operator

The comma is used in C and C++ not only as a separator in function argument lists, but also as an operator for sequential evaluation. The sole place that the comma *operator* is used in C# is in **for** loops, which will be described later in this chapter.

@Todo: Confirm that this is all the operators in C#

Common pitfalls when using operators

One of the pitfalls when using operators is trying to get away without parentheses when you are even the least bit uncertain about how an expression will evaluate. This is still true in C#.

An extremely common error in C and C++ looks like this:

```
while(x = y) {
    // ....
}
```

The programmer was trying to test for equivalence (==) rather than do an assignment. In C and C++ the result of this assignment will always be **true** if **y** is nonzero, and you'll probably get an infinite loop. In C#, the result of this expression is not a **bool,** and the compiler expects a **bool** and won't convert from an **int**, so it will conveniently give you a compile-time error and catch the problem before you ever try to run the program. So the pitfall never happens in C#. (The only time you won't get a compile-time error is when **x** and **y** are **bool**, in which case **x = y** is a legal expression, and in the above case, probably an error.)

A similar problem in C and C++ is using bitwise AND and OR instead of the logical versions. Bitwise AND and OR use one of the characters (**&** or |) while logical AND and OR use two (**&&** and ||). Just as with = and ==, it's easy to type just one character instead of two. In C#, the compiler again prevents this because it won't let you cavalierly use one type where it doesn't belong.

Casting operators

The word *cast* is used in the sense of "casting into a mold." C# will automatically change one type of data into another when appropriate. For instance, if you assign an integral value to a floating-point variable, the compiler will automatically convert the **int** to a **float**. Casting allows you to make this type conversion explicit, or to force it when it wouldn't normally happen.

To perform a cast, put the desired data type (including all modifiers) inside parentheses to the left of any value. Here's an example:

```
void casts() {
  int i = 200;
  long l = (long)i;
  long l2 = (long)200;
}
```

As you can see, it's possible to perform a cast on a numeric value as well as on a variable. In both casts shown here, however, the cast is superfluous, since the compiler will automatically promote an **int** value to a **long** when necessary. However, you are allowed to use superfluous casts to make a point or to make your code more clear. In other situations, a cast may be essential just to get the code to compile.

In C and C++, casting can cause some headaches. In C#, casting is safe, with the exception that when you perform a so-called *narrowing conversion* (that is, when you go from a data type that can hold more information to one that doesn't hold as much) you run the risk of losing information. Here the compiler forces you to do a cast, in effect saying "this can be a dangerous thing to do—if you want me to do it anyway you must make the cast explicit." With a *widening conversion* an explicit cast

is not needed because the new type will more than hold the information from the old type so that no information is ever lost.

Java allows you to cast any primitive type to any other primitive type, except for **bool,** which doesn't allow any casting at all. Class types do not allow casting. To convert one to the other there must be special methods. (You'll find out later in this book that objects can be cast within a *family* of types; an **Oak** can be cast to a **Tree** and vice-versa, but not to a foreign type such as a **Rock**.)

Literals

Ordinarily when you insert a literal value into a program the compiler knows exactly what type to make it. Sometimes, however, the type is ambiguous. When this happens you must guide the compiler by adding some extra information in the form of characters associated with the literal value. The following code shows these characters:

```
//: c03:Literals.cs
namespace c03{
    using System;

    public class Literals {
        //!char c = 0xffff; // max char hex value
        byte b = 0x7f; // max byte hex value
        short s = 0x7fff; // max short hex value
        int i1 = 0x2f; // Hexadecimal (lowercase)
        int i2 = 0X2F; // Hexadecimal (uppercase)
        int i3 = 0177; // Octal (leading zero)
        // Hex and Oct also work with long.
        long n1 = 200L; // long suffix
        long n2 = 200l; // long suffix - generates a
warning
        long n3 = 200;
        //! long l6(200); // not allowed
        float f1 = 1;
        float f2 = 1F; // float suffix
        float f3 = 1f; // float suffix
        float f4 = 1e-45f; // 10 to the power
        float f5 = 1e+9f; // float suffix
        double d1 = 1d; // double suffix
```

```
        double d2 = 1D; // double suffix
        double d3 = 47e47d; // 10 to the power
    }
}
//:~
@todo: Appropriate place for \u0081 types of things?
```

Hexadecimal (base 16), which works with all the integral data types, is denoted by a leading **0x** or **0X** followed by 0—9 and a—f either in upper or lowercase. If you try to initialize a variable with a value bigger than it can hold (regardless of the numerical form of the value), the compiler will give you an error message. Notice in the above code the maximum possible hexadecimal values for **char**, **byte,** and **short**. If you exceed these, the compiler will automatically make the value an **int** and tell you that you need a narrowing cast for the assignment. You'll know you've stepped over the line.

Octal (base 8) is denoted by a leading zero in the number and digits from 0-7. You'd think that, if they were going to provide octal (which I haven't seen used in a program in more than a decade), they'd provide a literal representation for binary numbers, but sadly, that is not the case.

A trailing character after a literal value establishes its type. Upper or lowercase **L** means **long**, upper or lowercase **F** means **float** and upper or lowercase **D** means **double**.

Exponents use a notation that some find rather dismaying: **1.39 e-47f**. In science and engineering, 'e' refers to the base of natural logarithms, approximately 2.718. (A more precise **double** value is available in C# as **Math.E**.) This is used in exponentiation expressions such as $1.39 \times e^{-47}$, which means 1.39×2.718^{-47}. However, when FORTRAN was invented they decided that **e** would naturally mean "ten to the power," which is an odd decision because FORTRAN was designed for science and engineering and one would think its designers would be sensitive about introducing such an ambiguity.[1] At any rate, this custom was followed in C, C++ and

[1] John Kirkham writes, "I started computing in 1962 using FORTRAN II on an IBM 1620. At that time, and throughout the 1960s and into the 1970s, FORTRAN was an all uppercase language. This probably started because many of the early input devices were old teletype units that used 5 bit Baudot code, which had no lowercase capability. The 'E'

now C#. So if you're used to thinking in terms of **e** as the base of natural logarithms, you must do a mental translation when you see an expression such as **1.39 e-47f** in Java; it means 1.39 x 10^{-47}.

Note that you don't need to use the trailing character when the compiler can figure out the appropriate type. With

```
long n3 = 200;
```

there's no ambiguity, so an **L** after the 200 would be superfluous. However, with

```
float f4 = 1e-47f; // 10 to the power
```

the compiler normally takes exponential numbers as doubles, so without the trailing **f** it will give you an error telling you that you must use a cast to convert **double** to **float**.

Promotion

You'll discover that if you perform any mathematical or bitwise operations on primitive data types that are smaller than an **int** (that is, **char**, **byte,** or **short**), those values will be promoted to **int** before performing the operations, and the resulting value will be of type **int**. So if you want to assign back into the smaller type, you must use a cast. (And, since you're assigning back into a smaller type, you might be losing information.) In general, the largest data type in an expression is the one that determines the size of the result of that expression; if you multiply a **float** and a **double**, the result will be **double**; if you add an **int** and a **long**, the result will be **long**.

---

in the exponential notation was also always upper case and was never confused with the natural logarithm base 'e', which is always lowercase. The 'E' simply stood for exponential, which was for the base of the number system used—usually 10. At the time octal was also widely used by programmers. Although I never saw it used, if I had seen an octal number in exponential notation I would have considered it to be base 8. The first time I remember seeing an exponential using a lowercase 'e' was in the late 1970s and I also found it confusing. The problem arose as lowercase crept into FORTRAN, not at its beginning. We actually had functions to use if you really wanted to use the natural logarithm base, but they were all uppercase."

C# has "sizeof"

The **sizeof( )** operator satisfies a specific need: it tells you the number of bytes allocated for data items. The most compelling need for **sizeof( )** in C and C++ is portability. Different data types might be different sizes on different machines, so the programmer must find out how big those types are when performing operations that are sensitive to size. For example, one computer might store integers in 32 bits, whereas another might store integers as 16 bits. Programs could store larger values in integers on the first machine. As you might imagine, portability is a huge headache for C and C++ programmers. In C#, this most common use of sizeof() is not relevant, but it can come into play when interfacing with external data structures or when you're manipulating blocks of raw data and you're willing to forego convenience and safety for every last bit of speed (say, if you're writing a routine for processing video data). The **sizeof()** operator is only usable inside unsafe code (see #).

C#'s Preprocessor

C#'s preprocessing directives should be used with caution.

MORE

These are *preprocessor directives* as described in chapter #preprocessing# and must be the only statements on a line. C# does not actually have a separate preprocessing step that runs prior to compilation but the form and use of these statements is intended to be familiar to C and C++ programmers.

While there's no harm in the #region directives, the other directives support *conditional compilation*, which allows a single codebase to generate multiple binary outputs. The most common use of conditional compilation is to remove debugging behavior from a shipping product; this is done by defining a symbol on the compilation command-line, and using the **#if**, **#endif**, **#else**, and **#elif** directives to create conditional logic depending on the existence or absence of one or more such symbols. Here's a simple example:

```
//:c13:CondComp.cs
//Demonstrates conditional compilation
```

```
class CondComp{
   public static void Main(){
#if DEBUG
      System.Console.WriteLine("Debug behavior");
#endif
   }
}
```

If **CondComp** is compiled with the command-line **csc /define:Debug CondComp.cs** it will print the line; if with a straight **csc CondComp.cs**, it won't. While this seems like a reasonable idea, in practice it often leads to situations where a change is made in one conditional branch and not in another, and the preprocessor leaves no trace in the code of the compilation options; in general, it's a better idea to use a **readonly bool** for such things. A reasonable compromise is to use the preprocessor directives to set the values of variables that are used to change runtime behavior:

```
//:c13:MarkedCondComp.cs
//Demonstrates conditional compilation

class CondComp{
   static readonly bool DEBUG =
#if DEBUG
   true;
#else
   false;
#endif

   public static void Main(){
      if(DEBUG)
          System.Console.WriteLine("Debug behavior");
   }
}///:~
```

In **MarkedCondComp**, if a problem arose, a debugger or logging facility would be able to read the value of the **DEBUG bool**, thus allowing the maintenance programmers to determine the compilation commands that lead to the troublesome behavior. The trivial disadvantages of this model are the slight penalty of a runtime comparison and the increase in the assembly's size due to the presence of the debugging code.

Precedence revisited

Operator precedence is difficult to remember, but here is a helpful mnemonic : "Ulcer Addicts Really Like C# A lot."

Mnemonic	Operator type	Operators
Ulcer	Unary	+ - ++--
Addicts	Arithmetic (and shift)	* / % + - << >>
Really	Relational	> < >= <= == !=
Like	Logical (and bitwise)	&& || & | ^
C#	Conditional (ternary)	A > B ? X : Y
A Lot	Assignment	= (and compound assignment like *=)

Of course, with the shift and bitwise operators distributed around the table it is not a perfect mnemonic, but for non-bit operations it works.

A compendium of operators

The following example shows which primitive data types can be used with particular operators. Basically, it is the same example repeated over and over, but using different primitive data types. The file will compile without error because the lines that would cause errors are commented out with a **//!**.

```
//:c03.AllOps.cs
namespace c03{
    using System;
// Tests all the operators on all the
// primitive data types to show which
// ones are accepted by the Java compiler.

public class AllOps {
  // To accept the results of a boolean test:
  void F(bool b) {}
  void boolTest(bool x, bool y) {
    // Arithmetic operators:
    //! x = x * y;
    //! x = x / y;
```

```
    //! x = x % y;
    //! x = x + y;
    //! x = x - y;
    //! x++;
    //! x--;
    //! x = +y;
    //! x = -y;
    // Relational and logical:
    //! F(x > y);
    //! F(x >= y);
    //! F(x < y);
    //! F(x <= y);
    F(x == y);
    F(x != y);
    F(!y);
    x = x && y;
    x = x || y;
    // Bitwise operators:
    //! x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Compound assignment:
    //! x += y;
    //! x -= y;
    //! x *= y;
    //! x /= y;
    //! x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! char c = (char)x;
    //! byte B = (byte)x;
    //! short s = (short)x;
```

```
    //! int i = (int)x;
    //! long l = (long)x;
    //! float f = (float)x;
    //! double d = (double)x;
  }
  void charTest(char x, char y) {
    // Arithmetic operators:
    x = (char)(x * y);
    x = (char)(x / y);
    x = (char)(x % y);
    x = (char)(x + y);
    x = (char)(x - y);
    x++;
    x--;
    x = (char)+y;
    x = (char)-y;
    // Relational and logical:
    F(x > y);
    F(x >= y);
    F(x < y);
    F(x <= y);
    F(x == y);
    F(x != y);
    //! F(!x);
    //! F(x && y);
    //! F(x || y);
    // Bitwise operators:
    x= (char)~y;
    x = (char)(x & y);
    x  = (char)(x | y);
    x = (char)(x ^ y);
    x = (char)(x << 1);
    x = (char)(x >> 1);
    //! @todo x = (char)(x >>> 1);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
```

```
    x >>= 1;
    //@todo x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! bool b = (bool)x;
    byte B = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void byteTest(byte x, byte y) {
    // Arithmetic operators:
    x = (byte)(x* y);
    x = (byte)(x / y);
    x = (byte)(x % y);
    x = (byte)(x + y);
    x = (byte)(x - y);
    x++;
    x--;
    x = (byte)+ y;
    x = (byte)- y;
    // Relational and logical:
    F(x > y);
    F(x >= y);
    F(x < y);
    F(x <= y);
    F(x == y);
    F(x != y);
    //! F(!x);
    //! F(x && y);
    //! F(x || y);
    // Bitwise operators:
    x = (byte)~y;
    x = (byte)(x & y);
    x = (byte)(x | y);
    x = (byte)(x ^ y);
    x = (byte)(x << 1);
```

```
    x = (byte)(x >> 1);
    //@todo x = (byte)(x >>> 1);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    //@todo x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! bool b = (bool)x;
    char c = (char)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void shortTest(short x, short y) {
    // Arithmetic operators:
    x = (short)(x * y);
    x = (short)(x / y);
    x = (short)(x % y);
    x = (short)(x + y);
    x = (short)(x - y);
    x++;
    x--;
    x = (short)+y;
    x = (short)-y;
    // Relational and logical:
    F(x > y);
    F(x >= y);
    F(x < y);
    F(x <= y);
    F(x == y);
    F(x != y);
```

```
    //! F(!x);
    //! F(x && y);
    //! F(x || y);
    // Bitwise operators:
    x = (short)~y;
    x = (short)(x & y);
    x = (short)(x | y);
    x = (short)(x ^ y);
    x = (short)(x << 1);
    x = (short)(x >> 1);
    //@todo x = (short)(x >>> 1);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    //@todo x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! bool b = (bool)x;
    char c = (char)x;
    byte B = (byte)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void intTest(int x, int y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
```

```
    x = +y;
    x = -y;
    // Relational and logical:
    F(x > y);
    F(x >= y);
    F(x < y);
    F(x <= y);
    F(x == y);
    F(x != y);
    //! F(!x);
    //! F(x && y);
    //! F(x || y);
    // Bitwise operators:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    //@todo x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    //@todo x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! bool b = (bool)x;
    char c = (char)x;
    byte B = (byte)x;
    short s = (short)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
```

```
void longTest(long x, long y) {
  // Arithmetic operators:
  x = x * y;
  x = x / y;
  x = x % y;
  x = x + y;
  x = x - y;
  x++;
  x--;
  x = +y;
  x = -y;
  // Relational and logical:
  F(x > y);
  F(x >= y);
  F(x < y);
  F(x <= y);
  F(x == y);
  F(x != y);
  //! F(!x);
  //! F(x && y);
  //! F(x || y);
  // Bitwise operators:
  x = ~y;
  x = x & y;
  x = x | y;
  x = x ^ y;
  x = x << 1;
  x = x >> 1;
  //@todo x = x >>> 1;
  // Compound assignment:
  x += y;
  x -= y;
  x *= y;
  x /= y;
  x %= y;
  x <<= 1;
  x >>= 1;
  //@todo x >>>= 1;
  x &= y;
  x ^= y;
  x |= y;
```

```
    // Casting:
    //! bool b = (bool)x;
    char c = (char)x;
    byte B = (byte)x;
    short s = (short)x;
    int i = (int)x;
    float f = (float)x;
    double d = (double)x;
  }
  void floatTest(float x, float y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    F(x > y);
    F(x >= y);
    F(x < y);
    F(x <= y);
    F(x == y);
    F(x != y);
    //! F(!x);
    //! F(x && y);
    //! F(x || y);
    // Bitwise operators:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
```

```
    x *= y;
    x /= y;
    x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    //! x &= y;
    //! x ^= y;
    //! x |= y;
    // Casting:
    //! bool b = (bool)x;
    char c = (char)x;
    byte B = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    double d = (double)x;
  }
  void doubleTest(double x, double y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    F(x > y);
    F(x >= y);
    F(x < y);
    F(x <= y);
    F(x == y);
    F(x != y);
    //! F(!x);
    //! F(x && y);
    //! F(x || y);
    // Bitwise operators:
    //! x = ~y;
```

```
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    //! x &= y;
    //! x ^= y;
    //! x |= y;
    // Casting:
    //! bool b = (bool)x;
    char c = (char)x;
    byte B = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
  }
}
}
//:~
```

Note that **bool** is quite limited. You can assign to it the values **true** and **false**, and you can test it for truth or falsehood, but you cannot add booleans or perform any other type of operation on them.

In **char**, **byte**, and **short** you can see the effect of promotion with the arithmetic operators. Each arithmetic operation on any of those types results in an **int** result, which must be explicitly cast back to the original type (a narrowing conversion that might lose information) to assign back to that type. With **int** values, however, you do not need to cast, because everything is already an **int**. Don't be lulled into thinking everything is

safe, though. If you multiply two **int**s that are big enough, you'll overflow the result. The following example demonstrates this:

```
//:c03:OverFlow.cs
namespace c03{
    using System;
public class Overflow {
  public static void Main() {
    int big = 0x7fffffff; // max int value
    Prt("big = " + big);
    int bigger = big * 4;
    Prt("bigger = " + bigger);
    bigger = checked(big * 4);
    //! Prt("never reached");
  }
  static void Prt(string s) {
    System.Console.WriteLine(s);
  }
}
}
//:~
```

The output of this is:

```
big = 2147483647
bigger = -4

Unhandled Exception: System.OverflowException:
Exception of type System.OverflowException was thrown.
   at c03.Overflow.Main()
```

If a potentially overflowing mathematical operation is not wrapped in the **checked()** keyword, you will get no errors or warnings from the compiler, and no exceptions at run-time. @todo: Question – why aren't all arithmetic operations in safe code checked and all arithmetic in unsafe code not?

Compound assignments do *not* require casts for **char**, **byte,** or **short**, even though they are performing promotions that have the same results

as the direct arithmetic operations. On the other hand, the lack of the cast certainly simplifies the code.

You can see that, with the exception of **boolean**, any primitive type can be cast to any other primitive type. Again, you must be aware of the effect of a narrowing conversion when casting to a smaller type, otherwise you might unknowingly lose information during the cast.

Execution control

C# uses all of C's execution control statements, so if you've programmed with C or C++ then most of what you see will be familiar. Most procedural programming languages have some kind of control statements, and there is often overlap among languages. In C#, the keywords include **if-else**, **while**, **do-while**, **for**, **foreach,** and a selection statement called **switch**. C# jumping keywords are **break, continue**, **goto** (yes, **goto**), and **return**.

true and false

All conditional statements use the truth or falsehood of a conditional expression to determine the execution path. An example of a conditional expression is **A == B**. This uses the conditional operator == to see if the value of **A** is equivalent to the value of **B**. The expression returns **true** or **false**. Any of the relational operators you've seen earlier in this chapter can be used to produce a conditional statement. Note that C# doesn't allow you to use a number as a **bool**, even though it's allowed in C and C++ (where truth is nonzero and falsehood is zero) and in Visual Basic (where truth is zero and falsehood non-zero). If you want to use a non-**bool** in a **bool** test, such as **if(a)**, you must first convert it to a **bool** value using a conditional expression, such as **if(a != 0)**.

if-else

The **if-else** statement is probably the most basic way to control program flow. The **else** is optional, so you can use **if** in two forms:

```
if(Boolean-expression)
  statement
```

or

```
if(Boolean-expression)
  statement
else
  statement
```

The conditional must produce a **bool** result. The *statement* means either a simple statement terminated by a semicolon or a compound statement, which is a group of simple statements enclosed in braces. Any time the word "*statement*" is used, it always implies that the statement can be simple or compound.

As an example of **if-else**, here is a **test( )** method that will tell you whether a guess is above, below, or equivalent to a target number:

```
//:c03:IfElse.cs
namespace c03{
    public class IfElse {
        static int Test(int testval, int target) {
            int result = 0;
            if (testval > target)
                result = +1;
            else if (testval < target)
                result = -1;
            else
                result = 0; // Match
            return result;
        }
        public static void Main() {
            System.Console.WriteLine(Test(10, 5));
            System.Console.WriteLine(Test(5, 10));
            System.Console.WriteLine(Test(5, 5));
        }
    }
}///:~
```

It is conventional to indent the body of a control flow statement so the reader might easily determine where it begins and ends.

Return

The **return** keyword has two purposes: it specifies what value a method will return (if it doesn't have a **void** return value) and it causes that value to be returned immediately. The **test( )** method above can be rewritten to take advantage of this:

```
//:c03:IfElse2.cs
    public class IfElse2 {
        static int Test(int testval, int target) {
            int result = 0;
            if (testval > target)
                return 1;
            else if (testval < target)
                return -1;
            else
                return 0; // Match
        }
        public static void Main() {
            System.Console.WriteLine(Test(10, 5));
            System.Console.WriteLine(Test(5, 10));
            System.Console.WriteLine(Test(5, 5));
        }
    }
///:~
```

Although this code has two **else**'s, they are actually unnecessary, because the method will not continue after executing a **return**. It is good programming practice to have as few exit points as possible in a method; a reader should be able to see at a glance the "shape" of a method without having to think "Oh! Unless something happens in this conditional, in which case it never gets to this other area.". After rewriting the method so that there's only one exit point, we can add extra functionality to the method and know that it will always be called:

```
//:c03:IfElse3.cs
public class IfElse3 {
    static int Test(int testval, int target) {
        int result = 0;  //Match
        if (testval > target)
```

```
            result = 1;
        else if (testval < target)
            result = -1;
        System.Console.WriteLine("All paths pass
here");
        return result;
    }
    public static void Main() {
        System.Console.WriteLine(Test(10, 5));
        System.Console.WriteLine(Test(5, 10));
        System.Console.WriteLine(Test(5, 5));
    }
} ///:~
```

Iteration

while, **do-while,** and **for** control looping and are sometimes classified as *iteration statements*. A *statement* repeats until the controlling *Boolean-expression* evaluates to false. The form for a **while** loop is

```
while(Boolean-expression)
  statement
```

The *Boolean-expression* is evaluated once at the beginning of the loop and again before each further iteration of the *statement*.

Here's a simple example that generates random numbers until a particular condition is met:

```
//:c03:WhileTest.cs
namespace c03{
    using System;
    // Demonstrates the while loop.
    public class WhileTest {
        public static void Main() {
            Random rand = new Random();
            double r = 0;
            while (r < 0.99d) {
                r = rand.NextDouble();
                System.Console.WriteLine(r);
            }
```

```
        }
    }
}///:~
```

This uses the **static** method **random( )** in the **Math** library, which generates a **double** value between 0 and 1. (It includes 0, but not 1.) The conditional expression for the **while** says "keep doing this loop until the number is 0.99 or greater." Each time you run this program you'll get a different-sized list of numbers.

do-while

The form for **do-while** is

```
do
  statement
while(Boolean-expression);
```

The sole difference between **while** and **do-while** is that the statement of the **do-while** always executes at least once, even if the expression evaluates to false the first time. In a **while**, if the conditional is false the first time the statement never executes. In practice, **do-while** is less common than **while**.

for

A **for** loop performs initialization before the first iteration. Then it performs conditional testing and, at the end of each iteration, some form of "stepping." The form of the **for** loop is:

```
for(initialization; Boolean-expression; step)
  statement
```

Any of the expressions *initialization*, *Boolean-expression* or *step* can be empty. The expression is tested before each iteration, and as soon as it evaluates to **false** execution will continue at the line following the **for** statement. At the end of each loop, the *step* executes.

for loops are usually used for "counting" tasks:

```
//:c03:ListCharacters.cs
```

```
namespace c03{
    using System;
    // Demonstrates "for" loop by listing
    // all the ASCII characters.
    public class ListCharacters {
        public static void Main() {
            for ( char c = (char) 0; c < (char) 128;
c++)
                if (c != 26 )  // ANSI Clear screen
                    System.Console.WriteLine(
                                        "value: " +
(int)c +
                                        " character: " +
c);
        }
    }
}///:~
```

Note that the variable **c** is defined at the point where it is used, inside the control expression of the **for** loop, rather than at the beginning of the block denoted by the open curly brace. The scope of **c** is the expression controlled by the **for**.

Traditional procedural languages like C require that all variables be defined at the beginning of a block so when the compiler creates a block it can allocate space for those variables. In C#, you can spread your variable declarations throughout the block, defining them at the point that you need them. This allows a more natural coding style and makes code easier to understand.

You can define multiple variables within a **for** statement, but they must be of the same type:

```
for(int i = 0, j = 1;
    i < 10 && j != 11;
    i++, j++)
  /* body of for loop */;
```

The **int** definition in the **for** statement covers both **i** and **j**. The ability to define variables in the control expression is limited to the **for** loop. You

cannot use this approach with any of the other selection or iteration statements.

foreach

C# has a specialized iteration operator called **foreach**. Unlike the others, foreach does not loop based on a boolean expression. Rather, it executes a block of code on each element in an array or other collection. The form for **foreach** is:

```
foreach(type loopVariable in collection){
   statement
}
```

The **foreach** statement is a terse way to specify the most common type of loop and does so without introducing potentially confusing index variables. Compare the clarity of **foreach** and **for** in this example:

```
//:c03:ForEach.cs
class ForEach {
    public static void Main() {
        string[] months = {"January", "February",
"March", "April"}; //etc
        string[] weeks = {"1st", "2nd", "3rd", "4th"};
        string[] days = {"Sunday", "Monday",
"Tuesday", "Wednesday"}; //etc

        foreach(string month in months)
           foreach(string week in weeks)
             foreach(string day in days)
                System.Console.WriteLine("{0} {1} week
{2}", month, week, day);

        for(int month = 0; month < months.Length;
month++)
            for(int week = 0; week < weeks.Length;
week++)
                for(int day = 0; day < days.Length;
day++)
```

```
                    System.Console.WriteLine("{0} {1}
week {2}", months[month], weeks[week], days[day]);

    }
} ///:~
```

Another advantage of **foreach** is that it performs an implicit typecast on objects stored in collections, saving a few more keystrokes when objects are stored not in arrays, but in more complex data structures. We'll cover this aspect of **foreach** in Chapter #.

The comma operator

Earlier in this chapter I stated that the comma *operator* (not the comma *separator*, which is used to separate definitions and function arguments) has only one use in C#: in the control expression of a **for** loop. In both the initialization and step portions of the control expression you can have a number of statements separated by commas, and those statements will be evaluated sequentially. The previous bit of code uses this ability. Here's another example:

```
//:c03:ListCharacters.cs
namespace c03{
    using System;
    public class CommaOperator {
  public static void Main() {
    for(int i = 1, j = i + 10; i < 5;
        i++, j = i * 2) {
      System.Console.WriteLine("i= " + i + " j= " +
j);
    }
  }
}
 }///:~
```

Here's the output:

```
i= 1 j= 11
i= 2 j= 4
i= 3 j= 6
i= 4 j= 8
```

You can see that in both the initialization and step portions the statements are evaluated in sequential order. Also, the initialization portion can have any number of definitions *of one type*.

break and continue

Inside the body of any of the iteration statements you can also control the flow of the loop by using **break** and **continue**. **break** quits the loop without executing the rest of the statements in the loop. **continue** stops the execution of the current iteration and goes back to the beginning of the loop to begin the next iteration.

This program shows examples of **break** and **continue** within **for** and **while** loops:

```
//:c03:BreakAndContinue.cs
namespace c03{
    using System;
    // Demonstrates break and continue keywords.

public class BreakAndContinue {
  public static void Main() {
//@todo can't declare I after loop!
    int i = 0;
    for(i = 0; i < 100; i++) {
      if(i == 74) break; // Out of for loop
      if(i % 9 != 0) continue; // Next iteration
      System.Console.WriteLine(i);
    }
    i = 0;
    // An "infinite loop":
    while(true) {
      i++;
      int j = i * 27;
      if(j == 1269) break; // Out of loop
      if(i % 10 != 0) continue; // Top of loop
      System.Console.WriteLine(i);
    }
  }
}
```

```
}///:~
```

In the **for** loop the value of **i** never gets to 100 because the **break** statement breaks out of the loop when **i** is 74. Normally, you'd use a **break** like this only if you didn't know when the terminating condition was going to occur. The **continue** statement causes execution to go back to the top of the iteration loop (thus incrementing **i**) whenever **i** is not evenly divisible by 9. When it is, the value is printed.

The second portion shows an "infinite loop" that would, in theory, continue forever. However, inside the loop there is a **break** statement that will break out of the loop. In addition, you'll see that the **continue** moves back to the top of the loop without completing the remainder. (Thus printing happens in the second loop only when the value of **i** is divisible by 10.) The output is:

```
0
9
18
27
36
45
54
63
72
10
20
30
40
```

The value 0 is printed because 0 % 9 produces 0.

A second form of the infinite loop is **for(;;)**. The compiler treats both **while(true)** and **for(;;)** in the same way, so whichever one you use is a matter of programming taste. (Although **while(true)** is clearly the only choice for a person of discriminating taste.)

The infamous "goto"

The **goto** keyword has been present in programming languages from the beginning. Indeed, **goto** was the genesis of program control in assembly

language: “if condition A, then jump here, otherwise jump there.” If you read the assembly code that is ultimately generated by virtually any compiler, you’ll see that program control contains many jumps. However, a **goto** is a jump at the source-code level, and that’s what brought it into disrepute. If a program will always jump from one point to another, isn’t there some way to reorganize the code so the flow of control is not so jumpy? **goto** fell into true disfavor with the publication of the famous 1968 paper “Go To Statement Considered Harmful” by Edsger Dijkstra (http://www.acm.org/classics/oct95/). Dijkstra argued that when you have a jump, the context that created the program state becomes difficult to visualize. Since then, goto-bashing has been a popular sport, with advocates of the cast-out keyword scurrying for cover.

As is typical in situations like this, the middle ground is the most fruitful. The problem is not the use of **goto**, but the overuse of **goto** or, indeed, any statement that makes it difficult to say “When this line is reached, the state of the system is necessarily such-and-such.” The best way to write code that makes system state easy to determine is to minimize cyclomatic complexity, which is a fancy way of saying “use as few selection and jump statements as possible.” Cyclomatic complexity is the measure of the number of possible paths through a block of code.

Error! Not a valid link.

In figure #, the methods simple() and alsoSimple() have a cyclomatic complexity of 1; there is only a single path through the code. It does not matter how long the method is, whether the method creates objects, or even if the method calls other, more complex, methods (if those methods have high complexity, so be it, it doesn’t affect the complexity of the method at hand). This simplicity is reflected in the control graph shown; a single line showing the direction of execution towards the exit point.

The method oneLoop() is slightly more complex. No matter what it’s input parameter, it will print out “Begin” and assign x to y at the very beginning of the for loop. That’s the first edge on its control graph (to help align with the source code, the figure shows a single “edge” as a straight length of code and a curving jump). Then, it *may* continue into the loop, assign z and print it, increment y, and loop; that’s the second edge. Finally, at some point, y will be equal to 10 and control will jump to the end of the

method. This is the third edge, as marked on the figure. Method twoExits() also has a cyclomatic complexity of 3, although its second edge doesn't loop, but exits.

The next method, twoLoops(), hardly seems more complex than oneLoop(), but if you look at its control graph, you can count five distinct edges. Finally, we see a visual representation of what programmer's call "spaghetti code." With a cyclomatic complexity of 12, spaghetti() is about as complex as a method should *ever* be. Once a method has more than about six conditional and iteration operators, it starts to become difficult to understand the ramifications of any changes. In the 1950s, the psychologist George Miller published a paper that said that "Seven plus or minus two" is the limit of our "span of absolute judgment." Trying to keep more than this number of things in our head at once is very error-prone. Luckily, we have this thing called "writing" (or, in our case, coding C#) which allows us to break the problem of "absolute judgment" into successive sub-problems, which can then be treated as units for the purpose of making higher-level judgments. Sounds like computer programming to me!

(The paper points out that by increasing the dimension of visual variables, we can achieve astonishing levels of discrimination as we do, say, when we recognize a friend we've not seen in years while rushing through an airport. It's interesting to note that computer programming hardly leverages this capacity at all. You can read the paper, which anticipates exactly the sort of thinking and choice-making common to programming, at http://www.well.com/user/smalin/miller.html)

In C#, **goto** can be used to jump within a method to a label. A label is an identifier followed by a colon, like this:

```
label1:
```

Although it's legal to place a label anywhere in a method, the only place where it's a good idea is right before an iteration statement. And the sole reason to put a label before an iteration is if you're going to nest another iteration or a switch inside it. That's because while **break** and **continue** interrupt only the loop that contains them, **goto** can interrupt the loops

up to where the label exists. Here is an example of the use and abuse of **goto**:

```
//:c03:Goto.cs
// Using Goto
using System;

public class Goto {
  public static void Main() {
    int i = 0;
     Random rand = new Random();
    outer: //Label before iterator
    for(; true ;) { // infinite loop
       prt("Prior to entering inner loop");
      inner: // Another label
      for(; i < 10; i++) {
        prt("i = " + i);
        if(i == 7) {
          prt("goto outer");
          i++; // Otherwise i never
               // gets incremented.
          goto outer;
        }
        if(i == 8) {
          prt("goto inner");
         i++; //Otherwise i never
                 //gets incremented
          goto inner;
        }
          double d = rand.NextDouble();
          if(i == 9 && d < 0.6){
                prt("Legal, but terrible");
                goto badIdea;
          }
         prt("Back in the loop");
       if(i == 9)
              goto bustOut;
      }
    }
    bustOut:
    prt("Exit loop");
```

```
      if(rand.NextDouble() < 0.5){
            goto spaghettiJump;
      }
     badIdea:
     prt("How did I get here?");
    goto outer;
     spaghettiJump:
    prt("Don't ever, ever do this.");
   }
   static void prt(string s) {
     System.Console.WriteLine(s);
   }
} ///:~
```

Things start out appropriately enough, with the labeling of the two loops as **outer** and **inner**. After counting to 7 and getting lulled into a false sense of security, control jumps out of both loops, and re-enters following the **outer** label. On the next loop, control jumps to the **inner** label. Then things get weird: if the random number generator comes up with a value less than 0.6, control jumps downwards, to the label marked **badIdea**, the method prints "How did I get here?" and then jumps all the way back to the **outer** label. On the next run through the inner loop, i is still equal to 9 but, eventually, the random number generator will come up with a value that will skip the jump to **badIdea** and print that we're "back in the loop." Then, instead of using the **for** statement's terminating condition, we decide that we're going to jump to the **bustOut** label. We do the programmatic equivalent of flipping a coin and either "fall through" into the **badIdea** area (which, of course, jumps us back to **outer**) or jump to the **spaghettiJump** label.

So why is this code considered so terrible? For one thing, it has a high cyclomatic complexity – it's just plain confusing. Also, note how much harder it is to understand program flow when one can't rely on brackets and indenting. And to make things worse, let's say you were debugging this code and you placed a breakpoint at the line `prt("How did I get here?")`. When the breakpoint is reached, there is no way, short of examining the output, for you to determine whether you reached it from the jump from the inner loop or from falling through from the immediately preceding lines (in this case, the program's output is

sufficient to this cause, but in the real world of complex systems, GUIs, and Web Services, it never is). As Dijkstra put it, "it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress."

By "coordinates" Dijkstra meant a way to know the path by which a system arrived in its current state. It's only with such a path in hand that one can debug, since challenging defects only become apparent sometime *after* the mistake has been made. (It is, of course, common to make mistakes immediately or just before the problem becomes apparent, but such mistakes aren't hard to root out and correct.) Dijkstra went on to say that his criticism was not just about **goto**, but that all language constructs "*should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.*" We'll revisit this concern when speaking of the way that C# and the .NET framework handle exceptions (obeying the requirement) and threading (which doesn't obey the requirement, but could).

Switch

The **switch** is sometimes classified as a *selection statement*. The **switch** statement selects from among pieces of code based on the value of an integral expression. Its form is:

```
switch(integral-selector) {
  case integral-value1 : statement; break;
  case integral-value2 : statement; return;
  case integral-value3 : statement; continue;
  case integral-value4 : statement; throw exception;
  case integral-value5 : statement; goto external-
label;
  case integral-value6 : //No statements
  case integral-value7 : statement; goto case
integral-value;
          // ...
  default: statement; break;
}
```

Integral-selector is an expression that produces an integral value. The **switch** compares the result of *integral-selector* to each *integral-value*. If it finds a match, the corresponding *statement* (simple or compound) executes. If no match occurs, the **default** *statement* executes.

You will notice in the above definition that each **case** ends with some kind of jump statement. The first one shown, **break**, is by far the most commonly used. Note that **goto** can be used in both the form discussed previously, which jumps to an arbitrary label in the enclosing statement block, and in a new form, **goto case**, which transfers control to the specified case block.

Unlike Java and C++, each case block, including the default block, must end in a jump statement. There is no "fall-through," although selectors may immediately precede other selectors. In the definition, this is seen at the selector for integral-value6, which will execute the statements in integral-value7's case block.

The **switch** statement is a clean way to implement multi-way selection (i.e., selecting from among a number of different execution paths), but it requires a selector that evaluates to a predefined type such as **int**, **char**, or **string**, or to an enumeration. For other types, you must use either a series of **if** statements, or implement some kind of conversion to one of the supported types. More generally, a well-designed object-oriented program will generally move a lot of control switching away from explicit tests in code into polymorphism (which we'll get to in #)

Here's an example that creates letters randomly and determines whether they're vowels or consonants:

```
//:c03: VowelsAndConsonants.cs
/// Demonstrates the switch statement.
namespace c03{
    using System;
public class VowelsAndConsonants {
  public static void Main() {
      Random rand = new Random();
    for(int i = 0; i < 100; i++) {
      char c = (char)(rand.Next('a','z'));
```

```
        System.Console.WriteLine(c + ": ");
        switch(c) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
                  System.Console.WriteLine("vowel");
                  break;
        case 'y':
        case 'w':
                  System.Console.WriteLine(
                    "Sometimes a vowel");
                  break;
        default:
                  System.Console.WriteLine("consonant");
                  break;
        }
      }
    }
}
}///:~
```

Since **char**s can be implicitly promoted to **int**s, Random.Next(int lowerBound, int upperBound) can be used to return values in the appropriate ASCII range.

Calculation details

THIS IS KIND OF INTERESTING STUFF BUT IS NOT RELEVANT TO THE ABOVE. MOVE ELSEWHERE? REWRITE TEXT

The statement:

```
char c = (char)(Math.random() * 26 + 'a');
```

deserves a closer look. **Math.random( )** produces a **double**, so the value 26 is converted to a **double** to perform the multiplication, which

also produces a **double**. This means that **'a'** must be converted to a **double** to perform the addition. The **double** result is turned back into a **char** with a cast.

What does the cast to **char** do? That is, if you have the value 29.7 and you cast it to a **char**, is the resulting value 30 or 29? The answer to this can be seen in this example:

```
//:c03:CastingNumbers.cs
namespace c03{
    using System;
// What happens when you cast a float
// or double to an integral value?

    public class CastingNumbers {
        public static void Main() {
            double
            above = 0.7,
            below = 0.4;
            System.Console.WriteLine("above: " +
above);
            System.Console.WriteLine("below: " +
below);
            System.Console.WriteLine(
                                    "(int)above: " +
(int)above);
            System.Console.WriteLine(
                                    "(int)below: " +
(int)below);
            System.Console.WriteLine(
                                    "(char)('a' +
above): " +
                                    (char)('a' +
above));
            System.Console.WriteLine(
                                    "(char)('a' +
below): " +
                                    (char)('a' +
below));
        }
    }
```

```
}///:~
```

The output is:

```
above: 0.7
below: 0.4
(int)above: 0
(int)below: 0
(char)('a' + above): a
(char)('a' + below): a
```

So the answer is that casting from a **float** or **double** to an integral value always truncates.

A second question concerns **Math.random( )**. Does it produce a value from zero to one, inclusive or exclusive of the value '1'? In math lingo, is it (0,1), or [0,1], or (0,1] or [0,1)? (The square bracket means "includes" whereas the parenthesis means "doesn't include.") Again, a test program might provide the answer:

```
//:c03:RandomBounds.cs
namespace c03{
    using System;
    // Does Random.Next() produce 0 and
Int32.Max_Value?

    public class RandomBounds {
        static void usage() {
            System.Console.WriteLine("Usage: \n\t" +
                                     "RandomBounds
lower\n\t" +
                                     "RandomBounds
upper\n\t" +
                                     "RandomBounds
bounded");
            Environment.Exit(1);
        }
        public static void Main(string[] args) {
            if (args.Length != 1) usage();
            Random rand = new Random();
            if (args[0] == "lower") {
                while (rand.Next() != 0)
                    ; // Keep trying
```

```
                System.Console.WriteLine("Produced
0!");
            } else if (args[0] == "upper") {
                while (rand.Next() != Int32.MaxValue)
                    ; // Keep trying
                System.Console.WriteLine("Produced " +
Int32.MaxValue);
            } else if (args[0] == "bounded") {
                bool foundLowerBound = false;
                bool foundUpperBound = false;
                do {
                    int bounded = rand.Next(0, 10);
                    if (bounded == 0) {

System.Console.WriteLine("Found lower bound!");
                        foundLowerBound = true;
                    }
                    if (bounded == 10) {

System.Console.WriteLine("Found upper bound!");
                        foundUpperBound = true;
                    }
                }while (!foundLowerBound ||
!foundUpperBound);
            } else
                usage();
        }
    }
} ///:~
```

To run the program, you type a command line of either:

```
RandomBounds lower
```

or

```
RandomBounds upper
```

In both cases you are forced to break out of the program manually, so it would appear that Math.random( ) never produces either 0.0 or 1.0. But

this is where such an experiment can be deceiving. If you consider[2] that there are about 2^{62} different double fractions between 0 and 1, the likelihood of reaching any one value experimentally might exceed the lifetime of one computer, or even one experimenter. It turns out that 0.0 is included in the output of Math.random( ). Or, in math lingo, it is [0,1).

Summary

This chapter concludes the study of fundamental features that appear in most programming languages: calculation, operator precedence, type casting, and selection and iteration. Now you're ready to begin taking steps that move you closer to the world of object-oriented programming. The next chapter will cover the important issues of initialization and cleanup of objects, followed in the subsequent chapter by the essential concept of implementation hiding.

Exercises

---

[2] Chuck Allison writes: The total number of numbers in a floating-point number system is
2(M-m+1)b^(p-1) + 1
where **b** is the base (usually 2), **p** is the precision (digits in the mantissa), **M** is the largest exponent, and **m** is the smallest exponent. IEEE 754 uses:
M = 1023, m = -1022, p = 53, b = 2
so the total number of numbers is
2(1023+1022+1)2^52
= 2((2^10-1) + (2^10-1))2^52
= (2^10-1)2^54
= 2^64 - 2^54
Half of these numbers (corresponding to exponents in the range [-1022, 0]) are less than 1 in magnitude (both positive and negative), so 1/4 of that expression, or 2^62 - 2^52 + 1 (approximately 2^62) is in the range [0,1). See my paper at http://www.freshsources.com/1995006a.htm (last of text).

5: Initialization & Cleanup

As the computer revolution progresses, "unsafe" programming has become one of the major culprits that makes programming expensive.

Two of these safety issues are *initialization* and *cleanup*. Many C bugs occur when the programmer forgets to initialize a variable. This is especially true with libraries when users don't know how to initialize a library component, or even that they must. Cleanup is a special problem because it's easy to forget about an element when you're done with it, since it no longer concerns you. Thus, the resources used by that element are retained and you can easily end up running out of resources (most notably, memory).

C++ introduced the concept of a *constructor* and a *destructor*, special methods automatically called when an object is created and destroyed. C# has these facilities, and in addition has a garbage collector that automatically releases memory resources when they're no longer being used. This chapter examines the issues of initialization and cleanup, and their support in C#.

Guaranteed initialization with the constructor

You can imagine creating a method called **Initialize( )** for every class you write. The name is a hint that it should be called before using the object. Unfortunately, this means the user must remember to call the method. In C#, the class designer can guarantee initialization of every object by providing a special method called a *constructor*. If a class has a constructor, C# automatically calls that constructor when an object is

created, before users can even get their hands on it. So initialization is guaranteed.

The next challenge is what to name this method. There are two issues. The first is that any name you use could clash with a name you might like to use as a member in the class. The second is that because the compiler is responsible for calling the constructor, it must always know which method to call. The C++ solution seems the easiest and most logical, so it's also used in C#: the name of the constructor is the same as the name of the class. It makes sense that such a method will be called automatically on initialization.

Here's a simple class with a constructor:

```
//:c04:SimpleConstructor.cs
namespace c04{
    using System;

// Demonstration of a simple constructor.
class Rock {
  public Rock() { // This is the constructor
    System.Console.WriteLine("Creating Rock");
  }
}

public class SimpleConstructor {
  public static void Main() {
    for(int i = 0; i < 10; i++)
      new Rock();
  }
}
}///:~
```

Now, when an object is created:

```
new Rock();
```

storage is allocated and the constructor is called. It is guaranteed that the object will be properly initialized before you can get your hands on it.

Note that the name of the constructor must match the name of the class *exactly*.

Like any method, the constructor can have arguments to allow you to specify *how* an object is created. The above example can easily be changed so the constructor takes an argument:

```
//:c04:SimpleConstrutor2.cs
namespace c04{
    using System;

// Demonstration of a simple constructor.
class Rock2 {
  public Rock2(int i) { // This is the constructor
    System.Console.WriteLine("Creating Rock number: "
+ i);
  }
}

public class SimpleConstructor {
  public static void Main() {
    for(int i = 0; i < 10; i++)
      new Rock2(i);
  }
}
}///:~
```

Constructor arguments provide you with a way to provide parameters for the initialization of an object. For example, if the class **Tree** has a constructor that takes a single integer argument denoting the height of the tree, you would create a **Tree** object like this:

```
Tree t = new Tree(12);  // 12-foot tree
```

If **Tree(int)** is your only constructor, then the compiler won't let you create a **Tree** object any other way.

Constructors eliminate a large class of problems and make the code easier to read. In the preceding code fragment, for example, you don't see an explicit call to some **initialize( )** method that is conceptually separate from definition. In C#, definition and initialization are unified concepts—you can't have one without the other.

The constructor is an unusual type of method because it has no return value. This is distinctly different from a **void** return value, in which the

method returns nothing but you still have the option to make it return something else. Constructors return nothing and you don't have an option. If there was a return value, and if you could select your own, the compiler would somehow need to know what to do with that return value. Accidentally typing a return type such as **void** before declaring a method is a common thing to do on a Monday morning, but the C# compiler won't allow it, telling you "member names cannot be the same as their enclosing type."

Method overloading

One of the important features in any programming language is the use of names. When you create an object, you give a name to a region of storage. A method is a name for an action. By using names to describe your system, you create a program that is easier for people to understand and change. It's a lot like writing prose—the goal is to communicate with your readers.

You refer to all objects and methods by using names. Well-chosen names make it easier for you and others to understand your code.

A problem arises when mapping the concept of nuance in human language onto a programming language. Often, the same word expresses a number of different meanings—it's *overloaded*. This is useful, especially when it comes to trivial differences. You say "wash the shirt," "wash the car," and "wash the dog." It would be silly to be forced to say, "shirtWash the shirt," "carWash the car," and "dogWash the dog" just so the listener doesn't need to make any distinction about the action performed. Most human languages are redundant, so even if you miss a few words, you can still determine the meaning. We don't need unique identifiers—we can deduce meaning from context.

Most programming languages (C in particular) require you to have a unique identifier for each function. So you could not have one function called **print( )** for printing integers and another called **print( )** for printing floats—each function requires a unique name.

In C# and other languages in the C++ family, another factor forces the overloading of method names: the constructor. Because the constructor's

name is predetermined by the name of the class, there can be only one constructor name. But what if you want to create an object in more than one way? For example, suppose you build a class that can initialize itself in a standard way or by reading information from a file. You need two constructors, one that takes no arguments (the *default* constructor, also called the *no-arg* constructor), and one that takes a **string** as an argument, which is the name of the file from which to initialize the object. Both are constructors, so they must have the same name—the name of the class. Thus, *method overloading* is essential to allow the same method name to be used with different argument types. And although method overloading is a must for constructors, it's a general convenience and can be used with any method.

Here's an example that shows both overloaded constructors and overloaded ordinary methods:

```
//:c04:OverLoading.cs
// Demonstration of both constructor
// and ordinary method overloading.
using System;

class Tree {
  int height;
  public Tree() {
    Prt("Planting a seedling");
    height = 0;
  }
  public Tree(int i) {
    Prt("Creating new Tree that is "
        + i + " feet tall");
    height = i;
  }
  internal void Info() {
    Prt("Tree is " + height
        + " feet tall");
  }
  internal void Info(string s) {
    Prt(s + ": Tree is "
        + height + " feet tall");
  }
```

```
  static void Prt(string s) {
    System.Console.WriteLine(s);
  }
}

public class Overloading {
  public static void Main() {
    for(int i = 0; i < 5; i++) {
      Tree t = new Tree(i);
      t.Info();
      t.Info("overloaded method");
    }
    // Overloaded constructor:
    new Tree();
  }
} ///:~
```

A **Tree** object can be created either as a seedling, with no argument, or as a plant grown in a nursery, with an existing height. To support this, there are two constructors, one that takes no arguments (we call constructors that take no arguments *default constructors*[1]) and one that takes the existing height.

You might also want to call the **info( )** method in more than one way. For example, with a **string** argument if you have an extra message you want printed, and without if you have nothing more to say. It would seem strange to give two separate names to what is obviously the same concept. Fortunately, method overloading allows you to use the same name for both.

Distinguishing overloaded methods

If the methods have the same name, how can C# know which method you mean? There's a simple rule: each overloaded method must take a unique list of argument types.

[1] Sometimes these constructors are described with the clumsy but thorough name "no-arg constructors." The term "default constructor" has been in use for many years and so I will use that.

If you think about this for a second, it makes sense: how else could a programmer tell the difference between two methods that have the same name, other than by the types of their arguments?

Even differences in the ordering of arguments are sufficient to distinguish two methods: (Although you don't normally want to take this approach, as it produces difficult-to-maintain code.)

```
//:c04:OverLoadingOrder.cs
// Overloading based on the order of
// the arguments.

public class OverloadingOrder {
  static void print(string s, int i) {
    System.Console.WriteLine(
      "string: " + s +
      ", int: " + i);
  }
  static void print(int i, string s) {
    System.Console.WriteLine(
      "int: " + i +
      ", string: " + s);
  }
  public static void Main() {
    print("string first", 11);
    print(99, "Int first");
  }
} ///:~
```

The two **print( )** methods have identical arguments, but the order is different, and that's what makes them distinct.

Overloading with primitives

A primitive can be automatically promoted from a smaller type to a larger one and this can be slightly confusing in combination with overloading. The following example demonstrates what happens when a primitive is handed to an overloaded method:

```
//:c04:PrimitiveOverloading.cs
// Promotion of primitives and overloading.
```

```
public class PrimitiveOverloading {
  // boolean can't be automatically converted
  static void Prt(string s) {
    System.Console.WriteLine(s);
  }

  void F1(char x) { Prt("F1(char)"); }
  void F1(byte x) { Prt("F1(byte)"); }
  void F1(short x) { Prt("F1(short)"); }
  void F1(int x) { Prt("F1(int)"); }
  void F1(long x) { Prt("F1(long)"); }
  void F1(float x) { Prt("F1(float)"); }
  void F1(double x) { Prt("F1(double)"); }

  void F2(byte x) { Prt("F2(byte)"); }
  void F2(short x) { Prt("F2(short)"); }
  void F2(int x) { Prt("F2(int)"); }
  void F2(long x) { Prt("F2(long)"); }
  void F2(float x) { Prt("F2(float)"); }
  void F2(double x) { Prt("F2(double)"); }

  void F3(short x) { Prt("F3(short)"); }
  void F3(int x) { Prt("F3(int)"); }
  void F3(long x) { Prt("F3(long)"); }
  void F3(float x) { Prt("F3(float)"); }
  void F3(double x) { Prt("F3(double)"); }

  void F4(int x) { Prt("F4(int)"); }
  void F4(long x) { Prt("F4(long)"); }
  void F4(float x) { Prt("F4(float)"); }
  void F4(double x) { Prt("F4(double)"); }

  void F5(long x) { Prt("F5(long)"); }
  void F5(float x) { Prt("F5(float)"); }
  void F5(double x) { Prt("F5(double)"); }

  void F6(float x) { Prt("F6(float)"); }
  void F6(double x) { Prt("F6(double)"); }

  void F7(double x) { Prt("F7(double)"); }
```

```
  void TestConstVal() {
    Prt("Testing with 5");
    F1(5);F2(5);F3(5);F4(5);F5(5);F6(5);F7(5);
  }
  void TestChar() {
    char x = 'x';
    Prt("char argument:");
    F1(x);F2(x);F3(x);F4(x);F5(x);F6(x);F7(x);
  }
  void TestByte() {
    byte x = 0;
    Prt("byte argument:");
    F1(x);F2(x);F3(x);F4(x);F5(x);F6(x);F7(x);
  }
  void TestShort() {
    short x = 0;
    Prt("short argument:");
    F1(x);F2(x);F3(x);F4(x);F5(x);F6(x);F7(x);
  }
  void TestInt() {
    int x = 0;
    Prt("int argument:");
    F1(x);F2(x);F3(x);F4(x);F5(x);F6(x);F7(x);
  }
  void TestLong() {
    long x = 0;
    Prt("long argument:");
    F1(x);F2(x);F3(x);F4(x);F5(x);F6(x);F7(x);
  }
  void TestFloat() {
    float x = 0;
    Prt("Float argument:");
    F1(x);F2(x);F3(x);F4(x);F5(x);F6(x);F7(x);
  }
  void TestDouble() {
    double x = 0;
    Prt("double argument:");
    F1(x);F2(x);F3(x);F4(x);F5(x);F6(x);F7(x);
  }
  public static void Main() {
    PrimitiveOverloading p =
```

```
      new PrimitiveOverloading();
    p.TestConstVal();
    p.TestChar();
    p.TestByte();
    p.TestShort();
    p.TestInt();
    p.TestLong();
    p.TestFloat();
    p.TestDouble();
  }
} ///:~
```

If you view the output of this program, you'll see that the constant value 5 is treated as an **int**, so if an overloaded method is available that takes an **int** it is used. In all other cases, if you have a data type that is smaller than the argument in the method, that data type is promoted. **char** produces a slightly different effect, since if it doesn't find an exact **char** match, it is promoted to **int**.

What happens if your argument is *bigger* than the argument expected by the overloaded method? A modification of the above program gives the answer:

```
//:c04:Demotion.cs
// Demotion of primitives and overloading.

public class Demotion {
  static void Prt(string s) {
    System.Console.WriteLine(s);
  }

  void F1(char x) { Prt("F1(char)"); }
  void F1(byte x) { Prt("F1(byte)"); }
  void F1(short x) { Prt("F1(short)"); }
  void F1(int x) { Prt("F1(int)"); }
  void F1(long x) { Prt("F1(long)"); }
  void F1(float x) { Prt("F1(float)"); }
  void F1(double x) { Prt("F1(double)"); }

  void F2(char x) { Prt("F2(char)"); }
  void F2(byte x) { Prt("F2(byte)"); }
  void F2(short x) { Prt("F2(short)"); }
```

```
  void F2(int x) { Prt("F2(int)"); }
  void F2(long x) { Prt("F2(long)"); }
  void F2(float x) { Prt("F2(float)"); }

  void F3(char x) { Prt("F3(char)"); }
  void F3(byte x) { Prt("F3(byte)"); }
  void F3(short x) { Prt("F3(short)"); }
  void F3(int x) { Prt("F3(int)"); }
  void F3(long x) { Prt("F3(long)"); }

  void F4(char x) { Prt("F4(char)"); }
  void F4(byte x) { Prt("F4(byte)"); }
  void F4(short x) { Prt("F4(short)"); }
  void F4(int x) { Prt("F4(int)"); }

  void F5(char x) { Prt("F5(char)"); }
  void F5(byte x) { Prt("F5(byte)"); }
  void F5(short x) { Prt("F5(short)"); }

  void F6(char x) { Prt("F6(char)"); }
  void F6(byte x) { Prt("F6(byte)"); }

  void F7(char x) { Prt("F7(char)"); }

  void TestDouble() {
    double x = 0;
    Prt("double argument:");
    F1(x);F2((float)x);F3((long)x);F4((int)x);
    F5((short)x);F6((byte)x);F7((char)x);
  }
  public static void Main() {
    Demotion p = new Demotion();
    p.TestDouble();
  }
} ///:~
```

Here, the methods take narrower primitive values. If your argument is wider then you must *cast* to the necessary type using the type name in parentheses. If you don't do this, the compiler will issue an error message.

You should be aware that this is a *narrowing conversion,* which means you might lose information during the cast. This is why the compiler forces you to do it—to flag the narrowing conversion.

Overloading on return values

It is common to wonder "Why only class names and method argument lists? Why not distinguish between methods based on their return values?" For example, these two methods, which have the same name and arguments, are easily distinguished from each other:

```
void f() {}
int f() {}
```

This works fine when the compiler can unequivocally determine the meaning from the context, as in **int x = f( )**. However, you can call a method and ignore the return value; this is often referred to as *calling a method for its side effect* since you don't care about the return value but instead want the other effects of the method call. So if you call the method this way:

```
f();
```

how can C# determine which **f( )** should be called? And how could someone reading the code see it? Because of this sort of problem, you cannot use return value types to distinguish overloaded methods.

Default constructors

As mentioned previously, a default constructor (a.k.a. a "no-arg" constructor) is one without arguments, used to create a "vanilla object." If you create a class that has no constructors, the compiler will automatically create a default constructor for you. For example:

```
//:c04:DefaultConstructor.cs
class Bird {
  int i;
}

public class DefaultConstructor {
  public static void Main() {
    Bird nc = new Bird(); // default!
```

```
  }
}///:~
```

The line

```
new Bird();
```

creates a new object and calls the default constructor, even though one was not explicitly defined. Without it we would have no method to call to build our object. However, if you define any constructors (with or without arguments), the compiler will *not* synthesize one for you:

```
class Bush {
  Bush(int i) {}
  Bush(double d) {}
}
```

Now if you say:

```
new Bush();
```

the compiler will complain that it cannot find a constructor that matches. It's as if when you don't put in any constructors, the compiler says "You are bound to need *some* constructor, so let me make one for you." But if you write a constructor, the compiler says "You've written a constructor so you know what you're doing; if you didn't put in a default it's because you meant to leave it out."

The **this** keyword

If you have two objects of the same type called **a** and **b**, you might wonder how it is that you can call a method **f( )** for both those objects:

```
class Banana { void f(int i) { /* ... */ } }
Banana a = new Banana(), b = new Banana();
a.f(1);
b.f(2);
```

If there's only one method called **f( )**, how can that method know whether it's being called for the object **a** or **b**?

To allow you to write the code in a convenient object-oriented syntax in which you "send a message to an object," the compiler does some undercover work for you. There's a secret first argument passed to the

method **f( )**, and that argument is the reference to the object that's being manipulated. So the two method calls above become something like:

```
Banana.f(a,1);
Banana.f(b,2);
```

This is internal and you can't write these expressions and get the compiler to interchange them with a.f()-style calls, but it gives you an idea of what's happening.

Suppose you're inside a method and you'd like to get the reference to the current object. Since that reference is passed *secretly* by the compiler, there's no identifier for it. However, for this purpose there's a keyword: **this**. The **this** keyword produces a reference to the object the method has been called for. You can treat this reference just like any other object reference. Keep in mind that if you're calling a method of your class from within another method of your class, you don't need to use **this**; you simply call the method. The current **this** reference is automatically used for the other method. Thus you can say:

```
class Apricot {
  int id;
  void pick() { /* ... */ }
  void pit() { pick(); id; /* ... */ }
}
```

Inside **pit( )**, you *could* say **this.pick( )** or **this.id** but there's no need to. The compiler does it for you automatically. The **this** keyword is used only for those special cases in which you need to explicitly use the reference to the current object. For example, it's often used in **return** statements when you want to return the reference to the current object:

```
//:c04:Leaf.cs

// Simple use of the "this" keyword.

public class Leaf {
  int i = 0;
  Leaf Increment() {
    i++;
    return this;
```

```
  }
  void Print() {
    System.Console.WriteLine("i = " + i);
  }
  public static void Main() {
    Leaf x = new Leaf();
    x.Increment().Increment().Increment().Print();
  }
} ///:~
```

Because **increment( )** returns the reference to the current object via the **this** keyword, multiple operations can easily be performed on the same object.

Another place where it's often used is to allow method parameters to have the same name as instance variables. Previously, we talked about the value of overloading methods so that the programmer only had to remember the one, most logical name. Similarly, the names of method parameters and the names of instance variables may also have a single logical name. C# allows you to use the **this** reference to disambiguate method variables (also called "stack variables") from instance variables. For clarity, you should use this capability only when the parameter is going to either be assigned to the instance variable (such as in a constructor) or when the parameter is to be compared against the instance variable. Method variables that have no correlation with same-named instance variables are a common source of lazy defects:

```
//:c04:Name.cs

class Name{
    string givenName;
    string surname;
    public Name(string givenName, string surname){
        this.givenName = givenName;
        this.surname = surname;
    }

    public bool perhapsRelated(string surname){
        return this.surname == surname;
    }
```

```
    public void printGivenName(){
        /* Legal, but unwise */
        string givenName = "method variable";
        System.Console.WriteLine("givenName is: " +
givenName);
        System.Console.WriteLine("this.givenName is: "
+ this.givenName);
    }

    public static void Main(){
        Name vanGogh = new Name("Vincent", "van
Gogh");
        vanGogh.printGivenName();
        bool b = vanGogh.perhapsRelated("van Gogh");
        if(b){
            System.Console.WriteLine("He's a van
Gogh.");
        }
    }
}///:~
```

In the constructor, the parameters **givenName** and **surname** are assigned to the similarly-named instance variables and this is quite appropriate – calling the parameters **inGivenName** and **inSurname** (or worse, using parameter names such as **firstName** or **lastName** that do not correspond to the instance variables) would require explaining in the documentation. The **perhapsRelated()** method shows the other appropriate use – the **surname** passed in is to be compared to the instance's **surname**.

Unfortunately, the usage in **printGivenName()** is also legal. Here, a variable called **givenName** is created on the stack; it has nothing to do with the instance variable also called **givenName**. It may be unlikely that someone would accidentally create a method variable called **givenName**, but you'd be amazed at how many **name**, **id**, and **flag**s I've seen in my day! It's another reason why meaningful variable names are important.

Sidebar: Stack, Instance, and Parameter Variables

Sometimes you'll see code where half the variables begin with underscores and half the variables don't:

```
foo = _bar;
```

The intent is to use the prefix to distinguish between method variables that are created on the stack and go out of scope as soon as the method exits and variables that have longer lifespans. I hate this idiom. For one thing, its origin had to do with visibility, not storage, and C# has explicit and infinitely better visibility specifiers. For another, it's used inconsistently – almost as many people use the underscores for stack variables as use them for instance variables.

Sometimes you see code that prepends an 'm' to member variables names:

```
foo = mBar;
```

I hate this a little less than I hate the underscores. This type of naming convention is an offshoot of a C naming idiom called "Hungarian notation," that prepended type information to a variable name (so strings would be **strFoo**). This is a great idea if you're programming in C and everyone who has programmed Windows has seen their share of variables starting with 'h', but the time for this naming convention has passed.

If you want to distinguish between method and instance variables, use **this**. It's object-oriented, descriptive, and explicit.

//end sidebar

Calling constructors from constructors

When you write several constructors for a class, there are times when you'd like to call one constructor from another to avoid duplicating code. In C#, you can specify that another constructor execute before the current constructor. You do this using the ':' operator and the **this** keyword.

Normally, when you say **this**, it is in the sense of "this object" or "the current object," and by itself it produces the reference to the current object. In a constructor name, a colon followed by the **this** keyword takes on a different meaning: it makes an explicit call to the constructor that matches the specified argument list. Thus you have a straightforward way to call other constructors:

```
//:c04:Flower.cs
// Calling constructors with ": this."
public class Flower {
  int petalCount = 0;
  string s = "null";
  Flower(int petals) {
    petalCount = petals;
    System.Console.WriteLine(
      "Constructor w/ int arg only, petalCount= "
      + petalCount);
  }
  Flower(string ss) {
    System.Console.WriteLine(
      "Constructor w/ string arg only, s=" + ss);
    s = ss;
  }
  Flower(string s, int petals) : this(petals){
//!    this(s); // Can't call two!
    this.s = s; // Another use of "this"
    System.Console.WriteLine("string & int args");
  }
  Flower() : this("hi", 47) {
    System.Console.WriteLine(
      "default constructor (no args)");
  }
  void Print() {
    System.Console.WriteLine(
      "petalCount = " + petalCount + " s = "+ s);
  }
  public static void Main() {
    Flower x = new Flower();
    x.Print();
  }
}//:~
```

The constructor **Flower(String s, int petals)** shows that, while you can call one constructor using **this**, you cannot call two. @todo: confirm

The meaning of **static**

With the **this** keyword in mind, you can more fully understand what it means to make a method **static**. It means that there is no **this** for that

particular method. You cannot call non-**static** methods from inside **static** methods (although the reverse is possible), and you can call a **static** method for the class itself, without any object. In fact, that's primarily what a **static** method is for. It's as if you're creating the equivalent of a global function (from C). Except global functions are not permitted in C#, and putting the **static** method inside a class allows it access to other **static** methods and **static** fields.

Some people argue that **static** methods are not object-oriented since they do have the semantics of a global function; with a **static** method you don't send a message to an object, since there's no **this**. This is probably a fair argument, and if you find yourself using a *lot* of static methods you should probably rethink your strategy. However, **static**s are pragmatic and there are times when you genuinely need them, so whether or not they are "proper OOP" should be left to the theoreticians. Indeed, even Smalltalk has the equivalent in its "class methods." @todo: Confirm that polymorphism discussion has non-polymorphism of static methods

Cleanup: finalization and garbage collection

Programmers know about the importance of initialization, but often forget the importance of cleanup. After all, who needs to clean up an **int**? But with libraries, simply "letting go" of an object once you're done with it is not always safe. Of course, C# has the garbage collector to reclaim the memory of objects that are no longer used. Now consider a very unusual case. Suppose your object allocates "special" memory without using **new**. The garbage collector knows only how to release memory allocated *with* **new**, so it won't know how to release the object's "special" memory. To handle this case, C# provides a method called a *destructor* that you can define for your class. The destructor, like the constructor, shares the class name, but is prefaced with a tilde:

```
class MyClass{
   public MyClass(){ //Constructor }
   public ~MyClass(){ //Destructor }
}
```

C++ programmers will find this syntax familiar, but this is actually a dangerous mimic – the C# destructor has vastly different semantics, as you'll see. Here's how it's *supposed* to work. When the garbage collector is ready to release the storage used for your object, it will first call the object's destructor, and only on the next garbage-collection pass will it reclaim the object's memory. So if you choose to use the destructor, it gives you the ability to perform some important cleanup *at the time of garbage collection*.

This is a potential programming pitfall because some programmers, especially C++ programmers, because in C++ *objects always get destroyed* (in a bug-free program), whereas in C# objects do not always get garbage-collected. Or, put another way:

> Clean up after yourself.

If you remember this, you will stay out of trouble. What it means is that if there is some activity that must be performed before you no longer need an object, you must perform that activity yourself. For example, suppose that you open a file and write stuff to it. If you don't explicitly close that file, it might never get properly flushed to the disk. If you put some kind of flush-and-close functionality inside your destructor, then if the object is garbage-collected, the file will be tidied up properly, but if the object is not collected, you can have an open file that may or may not be usable. So a second point to remember is:

> Your objects might not get garbage-collected.

You might find that the storage for an object never gets released because your program never nears the point of running out of storage. If your program completes and the garbage collector never gets around to releasing the storage for any of your objects, that storage will be returned to the operating system *en masse* as the program exits. This is a good thing, because garbage collection has some overhead, and if you never do it you never incur that expense. In practice, it appears that the garbage collector does, in fact, get called when a C# process ends, but this is explicitly not guaranteed by the CLR.

What are destructors for?

You might believe at this point that you should not use a destructor as a general-purpose cleanup method. What good is it?

A third point to remember is:

> Garbage collection is only about memory.

That is, the sole reason for the existence of the garbage collector is to recover memory that your program is no longer using. So any activity that is associated with garbage collection, most notably your destructor method, must also be only about memory and its deallocation. Valuable resources, such as file handles, database connections, and sockets ought to be managed explicitly in your code, without relying on destructors. @todo: add section (later in chapter) about factory methods / singletons / pools

Does this mean that if your object contains other objects, your destructor should explicitly release those objects? Well, no—the garbage collector takes care of the release of all object memory regardless of how the object is created. It turns out that the need for destructors is limited to special cases, in which your object can allocate some storage in some way other than creating an object. But, you might observe, everything in C# is an object so how can this be?

It would seem that C# has a destructor because of its support for unmanaged code, in which you can allocate memory in a C-like manner. Memory allocated in unmanaged code is not restored by the garbage collection mechanism. The argument is probably that, for this special case, there should be symmetry in the language design and the destructor is the complement to the constructor. This is a weak argument. RAM may no longer be as scarce a commodity as it was in the past (when 64K was enough for anyone), but sloppiness with unmanaged memory is a huge practical problem, especially in these days of buffer overrun security attacks and the desire for machines to go weeks or months without rebooting.

After reading this, you probably get the idea that you won't be writing destructors too often. Good.

Instead of a destructor, use Close() or Dispose()

The majority of objects don't use resources that need to be cleaned up. So most of the time, you don't worry about what happens when they "go away." But if you do use a resource, you should write a method called **Close()** if the resource continues to exist after your use of it ends or **Dispose()** otherwise. Most importantly, you should *explicitly* call the **Close()** or **Dispose()** method as soon as you no longer require the resource.

If you rely on the garbage collector to manage resources, you can count on trouble:

```
//:c04:ValuableResource.cs
using System;
using System.Threading;

class ValuableResource{
   public static void Main(){
      useValuableResources();
      System.Console.WriteLine("Valuable resources
used and discarded");
      Thread.Sleep(10000);
      System.Console.WriteLine("10 seconds later...");
      //You would _think_ this would be fine
      ValuableResource vr = new ValuableResource();
  }

    static void useValuableResources(){
        for(int i = 0; i < MAX_RESOURCES; i++){
            ValuableResource vr = new
ValuableResource();
        }
    }

    static int idCounter;
    static int MAX_RESOURCES = 10;
    static int INVALID_ID = -1;
    int id;
```

```
    ValuableResource(){
        if(idCounter == MAX_RESOURCES){
            System.Console.WriteLine("No resources
available");
            id = INVALID_ID;
        }else{
            id = idCounter++;
            System.Console.WriteLine("Resource[{0}]
Constructed", id);
        }
    }
    ~ValuableResource(){
        if(id == INVALID_ID){
            System.Console.WriteLine("Things are
awry!");
        }else{
        idCounter--;
          System.Console.WriteLine("Resource[{0}]
Destructed", id );
        }
    }

}///:~
```

In this example, the first thing that happens upon entering Main() is the useValuableResources() method is called. This is straightforward – the MAX_RESOURCES number of ValuableResource objects are created and then immediately allowed to "go away". In the ValuableResource() constructor, the static idCounter variable is checked to see if it equals the MAX_RESOURCES value. If so, a "No resources available" message is written and the id of the Valuable Resource is set to an invalid value (in this case, the idCounter is the source of the "scarce" resource which is "consumed" by the id variable). The ValuableResource destructor either outputs a warning message or decrements the idCounter (thus, making another "resource" available).

When useValuableResources() returns, the system pauses for 10 seconds (we'll discuss Thread.Sleep() in great detail in Chapter #MULTITHREADING#), and finally a new ValuableResource is created. It seems like that should be fine, since those created in

useValuableResources() are long gone. But the output tells a different story:

```
Resource[0] Constructed
Resource[1] Constructed
Resource[2] Constructed
Resource[3] Constructed
Resource[4] Constructed
Resource[5] Constructed
Resource[6] Constructed
Resource[7] Constructed
Resource[8] Constructed
Resource[9] Constructed
Valuable resources used and discarded
10 seconds later...
No resources available
Things are awry!
Resource[9] Destructed
Resource[8] Destructed
Resource[7] Destructed
Resource[6] Destructed
Resource[5] Destructed
Resource[4] Destructed
Resource[3] Destructed
Resource[2] Destructed
Resource[1] Destructed
Resource[0] Destructed
```

Even after ten seconds (an eternity in computing time), no id's are available and the final attempt to create a ValuableResource fails. Interestingly, the Main() exits immediately after the "No resources available!" message is written. In this case, the CLR *did* do a garbage collection as the program exited and the ~ValuableResource() destructors got called. In this case, they happen to be deleted in the reverse order of their creation, but the order of destruction of resources is yet another "absolutely not guaranteed" characteristic of garbage collection.

Worse, this is the output if one presses Ctl-C during the pause:

```
Resource[0] Constructed
Resource[1] Constructed
Resource[2] Constructed
```

```
Resource[3] Constructed
Resource[4] Constructed
Resource[5] Constructed
Resource[6] Constructed
Resource[7] Constructed
Resource[8] Constructed
Resource[9] Constructed
Valuable resources used and discarded
^C
D:\tic\chap4>
```

That's it. No cleanup. If the valuable resources were, say, network sockets or database connections or files or, well, anything that actually had any value, they'd be lost until you reboot (or some other process manages to restore their state by brute force, as can happen with files).

```
//:c04:ValuableResource2.cs
using System;
using System.Threading;

class ValuableResource{
    static int idCounter;
    static int MAX_RESOURCES = 10;
    static int INVALID_ID = -1;
    int id;
    ValuableResource(){
        if(idCounter == MAX_RESOURCES){
            System.Console.WriteLine("No resources
available");
            id = INVALID_ID;
        }else{
            id = idCounter++;
            System.Console.WriteLine("Resource[{0}]
Constructed", id);
        }
    }
    void Dispose(){
        idCounter--;
        System.Console.WriteLine("Resource[{0}]
Destructed", id );
        if(id == INVALID_ID){
```

```
            System.Console.WriteLine("Things are
awry!");
        }
        GC.SuppressFinalize(this);
    }
    ~ValuableResource(){
        this.Dispose();
    }

    public static void Main(){
        useValuableResources();
        System.Console.WriteLine("Valuable resources
used and discarded");
        Thread.Sleep(10000);
        System.Console.WriteLine("10 seconds
later...");
        //This _is_ fine
        ValuableResource vr = new ValuableResource();
    }

    static void useValuableResources(){
        for(int i = 0; i < MAX_RESOURCES; i++){
            ValuableResource vr = new
ValuableResource();
            vr.Dispose();
        }
    }
}///:~
```

We've moved the code that was previously in the destructor into a method called **Dispose()**. Additionally, we've added the line:

```
GC.SuppressFinalize(this);
```

Which tells the Garbage Collector (the GC object) not to call the destructor during garbage collection. We've kept the destructor, but it does nothing but call **Dispose()**. In this case, the destructor is just a safety-net. It remains our responsibility to explicitly call Dispose(), but if we don't and it so happens that the garbage collector gets first up, then our bacon is pulled out of the fire. Some argue this is worse than useless -- a method which isn't guaranteed to be called but which performs a critical function.

When ValuableResources2 is run, not only are there no problems with running out of resources, the idCounter never gets above 0!

Destructors, Dispose(), and the using keyword

What follows is confusing. We've talked about the only valid use of the destructor being the guarantee that a valuable resource that exists as an instance variable will be cleaned up. The C# language has a way to guarantee that a special cleanup method is called, even if something unusual happens. What's confusing is that the technique calls *not* the destructor, but the **Dispose()** method! The technique uses object-oriented inheritance, which won't be discussed until chapter #inheritance#. Further, to illustrate it, I need to throw an Exception, a technique which won't be discussed until chapter #exceptions#! Rather than put off the discussion, though, I think it's important to present the technique here.

To ensure that a "clean up method" is called:

1. Inherit from IDisposable
2. Implement **public void Dispose()**
3. Place the vulnerable object inside a **using()** block

The **Dispose()** method will be called on exit from the **using** block. I'm not going to go over this example in detail, since it uses so many as-yet-unexplored features, but the key is the block that follows the **using()** declaration. When you run this code, you'll see that the **Dispose()** method is called, but not the destructor!

```
//:c04:UsingCleanup.cs
using System;

class UsingCleanup : IDisposable{
    public static void Main(){
       using(UsingCleanup uc = new UsingCleanup()){
             throw new NotImplementedException();
        }
```

```
    }

    UsingCleanup(){
        System.Console.WriteLine("Constructor
called");
    }

    public void Dispose(){
        System.Console.WriteLine("Dispose called");
    }

    ~UsingCleanup(){
        System.Console.WriteLine("Destructor called");
    }
}
//:~
```

One of the things **Dispose( )** can be useful for is probing the sometimes mysterious process of garbage collection.

@todo: Write a new program that illustrates better C#'s destructor / Dispose issues

The following example shows you what's going on and summarizes the previous descriptions of garbage collection:

```
//:c04:Garbage.cs

// Demonstration of the garbage
// collector and disposal

class Chair {
  static bool gcrun = false;
  internal static bool f = false;
  internal static int created = 0;
  internal static int finalized = 0;
  int i;
  internal Chair() {
    i = ++created;
    if(created == 47){
      System.Console.WriteLine("Created 47");
```

```
      System.Console.WriteLine("Bytes of memory
allocated before GC: " +
System.GC.GetTotalMemory(false));
      System.Console.WriteLine("Bytes of memory
allocated after GC: " +
System.GC.GetTotalMemory(true));
    }
  }
  ~Chair() {
    if(!gcrun) {
      // The first time the destructor is called:
      gcrun = true;
      System.Console.WriteLine(
        "Beginning to destruct after " +
        created + " Chairs have been created");
    }
    if(i == 47) {
      System.Console.WriteLine(
        "Destroying Chair #47, " +
        "Setting flag to stop Chair creation");
      f = true;
    }
    finalized++;
    if(finalized >= created)
      System.Console.WriteLine(
        "All " + finalized + " finalized");
  }
}

public class Garbage {
  public static void Main(string[] args) {
      System.Console.WriteLine("Bytes of memory
allocated: " + System.GC.GetTotalMemory(false));
    // As long as the flag hasn't been set,
    // make Chairs:
    while(!Chair.f) {
      new Chair();
    }
    System.Console.WriteLine(
      "After all Chairs have been created:\n" +
      "total created = " + Chair.created +
```

```
      ", total finalized = " + Chair.finalized);
    // Optional arguments force garbage
    // collection & finalization:
    if(args.Length > 0) {
      if(args[0] == "gc" ||
         args[0] == "all") {
        System.Console.WriteLine("gc():");
        System.GC.Collect();
      }
      /* @todo: Find equivalent?
      if(args[0].equals("finalize") ||
         args[0].equals("all")) {

System.Control.WriteLine("runFinalization():");
        System.runFinalization();
      }
    }
    */
    System.Console.WriteLine("bye!");
  }
}
}///:~
```

The above program creates many **Chair** objects, and at some point after the garbage collector begins running, the program stops creating **Chair**s. Since the garbage collector can run at any time, you don't know exactly when it will start up, so there's a flag called **gcrun** to indicate whether the garbage collector has started running yet. A second flag **f** is a way for **Chair** to tell the **Main( )** loop that it should stop making objects. Both of these flags are set within **~Chair( )**, which is called during garbage collection.

Two other **static** variables, **created** and **finalized**, keep track of the number of **Chair**s created versus the number that get finalized by the garbage collector. Finally, each **Chair** has its own (non-**static**) **int i** so it can keep track of what number it is. When **Chair** number 47 is finalized, the flag is set to **true** to bring the process of **Chair** creation to a stop.

All this happens in **Main( )**, in the loop

```
    while(!Chair.f) {
      new Chair();
```

```
    }
```

You might wonder how this loop could ever finish, since there's nothing inside the loop that changes the value of **Chair.f**. However, the **finalize( )** process will, eventually, when it finalizes number 47.

When you run the program, you provide a command-line argument of "gc," "finalize," or "all." The "gc" argument will call the **System.gc( )** method (to force execution of the garbage collector). Using the "finalize" argument calls **System.runFinalization( )** which—in theory—will cause any unfinalized objects to be finalized. And "all" causes both methods to be called.

The preceding program shows that the promise that finalizers will always be run holds true, but only if you explicitly force it to happen yourself. If you don't cause **System.gc( )** to be called, you'll get an output like this:

```
Created 47
Beginning to finalize after 3486 Chairs have been
created
Finalizing Chair #47, Setting flag to stop Chair
creation
After all Chairs have been created:
total created = 3881, total finalized = 2684
bye!
```

Thus, not all finalizers get called by the time the program completes. If **System.gc( )** is called, it will finalize and destroy all the objects that are no longer in use up to that point.

Remember that neither garbage collection nor finalization is guaranteed. If the Java Virtual Machine (JVM) isn't close to running out of memory, then it will (wisely) not waste time recovering memory through garbage collection.

The death condition

There is a very interesting use of the destructor, even if it's not called every time. That use is the verification of the *death condition*[2] of an object.

At the point that you're no longer interested in an object—when it's ready to be cleaned up—that object should be in a state whereby its memory can be safely released. For example, if the object represents an open file, that file should be closed by the programmer before the object is garbage-collected. If any portions of the object are not properly cleaned up, then you have a bug in your program that could be very difficult to find. The value of the destructors is that it can be used to discover this condition, even if the destructor isn't always called. If one of the destructors happens to reveal the bug, then you discover the problem, which is all you really care about.

Here's a simple example of how you might use it:

```
//:c04:DeathCondition.cs
// Using a destructor to detect an object that
// hasn't been properly cleaned up.

class Book {
  bool checkedOut = false;
  internal Book(bool checkOut) {
    checkedOut = checkOut;
  }
  internal void CheckIn() {
    checkedOut = false;
  }
  ~Book() {
    if(checkedOut)
      System.Console.WriteLine("Error: checked out");
  }
}
```

[2] A term coined by Bill Venners (www.artima.com)

```
public class DeathCondition {
  public static void Main() {
    Book novel = new Book(true);
    // Proper cleanup:
    novel.CheckIn();
    // Drop the reference, forget to clean up:
    new Book(true);
    // Force garbage collection & finalization:
    System.GC.Collect();
  }
} ///:~
```

The death condition is that all **Book** objects are supposed to be checked in before they are garbage-collected, but in **Main( )** a programmer error doesn't check in one of the books. Without the destructor's validation of the death condition, this could be a difficult bug to find.

Note that **System.GC.Collect( )** is used to force cleanup (and you should do this during program development to speed debugging). But even if it isn't, the garbage collector is always fired on the end of an application process (unless broken into with an explicit "Control-C" interrupt), so the errant **Book** will quickly be discovered.

How a garbage collector works

If you come from a programming language where allocating objects on the heap is expensive, you may naturally assume that C#'s scheme of allocating all reference types on the heap is expensive. However, it turns out that the garbage collector can have a significant impact on *increasing* the speed of object creation. This might sound a bit odd at first—that storage release affects storage allocation—but it means that allocating storage for heap objects in C# can be nearly as fast as creating storage on the stack in other languages.

For example, you can think of the C++ heap as a yard where each object stakes out its own piece of turf. This real estate can become abandoned sometime later and must be reused. In C#, the managed heap is quite different; it's more like a conveyor belt that moves forward every time you allocate a new object. This means that object storage allocation is remarkably rapid. The "heap pointer" is simply moved forward into virgin territory, so it's effectively the same as C++'s stack allocation. (Of course,

there's a little extra overhead for bookkeeping but it's nothing like searching for storage.) Yes, you heard right – allocation on the managed heap is *faster* than allocation within a C++-style unmanaged heap.

Now you might observe that the heap isn't in fact a conveyor belt, and if you treat it that way you'll eventually start paging memory a lot (which is a big performance hit) and later run out. The trick is that the garbage collector steps in and while it collects the garbage it compacts all the objects in the heap so that you've effectively moved the "heap pointer" closer to the beginning of the conveyor belt and further away from a page fault. The garbage collector rearranges things and makes it possible for the high-speed, infinite-free-heap model to be used while allocating storage.

To understand how this works, you need to get a little better idea of the way the Common Language Runtime garbage collector (GC) works. Garbage collection in the CLR (remember that memory management exists in the CLR "below" the level of the Common Type System, so this discussion equally applies to programs written in Visual Basic .NET, Eiffel .NET, and Python .NET as to C# programs) is based on the idea that any nondead object must ultimately be traceable back to a reference that lives either on the stack or in static storage. The chain might go through several layers of objects. Thus, if you start in the stack and the static storage area and walk through all the references you'll find all the live objects. For each reference that you find, you must trace into the object that it points to and then follow all the references in *that* object, tracing into the objects they point to, etc., until you've moved through the entire web that originated with the reference on the stack or in static storage. Each object that you move through must still be alive. Note that there is no problem with detached self-referential groups—these are simply not found, and are therefore automatically garbage. Also, if you trace to an object that has already been walked to, you do not have to re-trace it.

Having located all the "live" objects, the GC starts at the end of the managed heap and shifts the first live object in memory to be directly adjacent to the penultimate live object. This pair of live objects is then shifted to the next live object, the three are shifted en masse to the next, and so forth, until the heap is compacted.

Obviously, garbage collection is a lot of work, even on a modern, high-speed machine. In order to improve performance, the garbage collector refines the basic approach described here with *generations*.

The basic concept of generational garbage collection is that an object allocated recently is more likely to be garbage than an object which has already survived multiple passes of the garbage collector. So instead of walking the heap all the way from the stack or static storage, once the GC has run once, the collector may assume that the previously compacted objects (the older generation) are all valid and only walk the most recently allocated part of the heap (the new generation).

Garbage collection is a favorite topic of researchers, and there will undoubtedly be innovations in GC that will eventually find their way into the field. However, garbage collection and computer power have already gotten to the stage where the most remarkable thing about GC is how transparent it is.

Member initialization

C# goes out of its way to guarantee that variables are properly initialized before they are used. In the case of variables that are defined locally to a method, this guarantee comes in the form of a compile-time error. So if you say:

```
void F() {
  int i;
  i++;
}
```

you'll get an error message that says that **i** is an unassigned local variable. Of course, the compiler could have given **i** a default value, but it's more likely that this is a programmer error and a default value would have covered that up. Forcing the programmer to provide an initialization value is more likely to catch a bug.

If a primitive is a data member of a class, however, things are a bit different. Since any method can initialize or use that data, it might not be practical to force the user to initialize it to its appropriate value before the data is used. However, it's unsafe to leave it with a garbage value, so each

primitive data member of a class is guaranteed to get an initial value. Those values can be seen here:

```
//:c04:InitialValues.cs
// Shows default initial values.
class Measurement {
  bool t;
  char c;
  byte b;
  short s;
  int i;
  long l;
  float f;
  double d;
  internal void Print() {
    System.Console.WriteLine(
      "Data type      Initial value\n" +
      "bool           " + t + "\n" +
      "char           [" + c + "] "+ (int)c +"\n"+
      "byte           " + b + "\n" +
      "short          " + s + "\n" +
      "int            " + i + "\n" +
      "long           " + l + "\n" +
      "float          " + f + "\n" +
      "double         " + d);
  }
}

public class InitialValues {
  public static void Main() {
    Measurement d = new Measurement();
    d.Print();
    /* In this case you could also say:
    new Measurement().print();
    */
  }
} ///:~
```

The output of this program is:

```
Data type      Initial value
boolean        false
```

```
char            [ ] 0
byte            0
short           0
int             0
long            0
float           0.0
double          0.0
```

The **char** value is a zero, which prints as a space.

You'll see later that when you define an object reference inside a class without initializing it to a new object, that reference is given a special value of **null** (which is a C# keyword).

You can see that even though the values are not specified, they automatically get initialized. So at least there's no threat of working with uninitialized variables.

Specifying initialization

What happens if you want to give a variable an initial value? One direct way to do this is simply to assign the value at the point you define the variable in the class. Here the field definitions in class **Measurement** are changed to provide initial values:

```
class Measurement {
  bool b = true;
  char c = 'x';
  byte B = 47;
  short s = 0xff;
  int i = 999;
  long l = 1;
  float f = 3.14f;
  double d = 3.14159;
  //. . .
```

You can also initialize nonprimitive objects in this same way. If **Depth** is a class, you can insert a variable and initialize it like so:

```
class Measurement {
  Depth o = new Depth();
  boolean b = true;
```

```
  // . . .
```

If you haven't given **o** an initial value and you try to use it anyway, you'll get a run-time error called an *exception* (covered in Chapter #exception#).

You can even call a static method to provide an initialization value:

```
class CInit {
  int i = InitI();
  //...
  static int InitI(){ //… }
}
```

This method can have arguments, but those arguments cannot be instance variables. Java programmers will note that this is more restrictive than Java's instance initialization, which can call non-static methods and use previously instantiated instance variables.

This approach to initialization is simple and straightforward. It has the limitation that *every* object of type **Measurement** will get these same initialization values. Sometimes this is exactly what you need, but at other times you need more flexibility.

Constructor initialization

The constructor can be used to perform initialization, and this gives you greater flexibility in your programming since you can call methods and perform actions at run-time to determine the initial values. There's one thing to keep in mind, however: you aren't precluding the automatic initialization, which happens before the constructor is entered. So, for example, if you say:

```
class Counter {
  int i;
  Counter() { i = 7; }
  // . . .
```

then **i** will first be initialized to 0, then to 7. This is true with all the primitive types and with object references, including those that are given explicit initialization at the point of definition. For this reason, the compiler doesn't try to force you to initialize elements in the constructor

at any particular place, or before they are used—initialization is already guaranteed[3].

Order of initialization

Within a class, the order of initialization is determined by the order that the variables are defined within the class. The variable definitions may be scattered throughout and in between method definitions, but the variables are initialized before any methods can be called—even the constructor. For example:

```
//:c04:OrderOfInitialization.cs
// Demonstrates initialization order.

// When the constructor is called to create a
// Tag object, you'll see a message:
class Tag {
  internal Tag(int marker) {
    System.Console.WriteLine("Tag(" + marker + ")");
  }
}

class Card {
  Tag t1 = new Tag(1); // Before constructor
  internal Card() {
    // Indicate we're in the constructor:
    System.Console.WriteLine("Card()");
    t3 = new Tag(33); // Reinitialize t3
  }
  Tag t2 = new Tag(2); // After constructor
  internal void F() {
    System.Console.WriteLine("F()");
  }
  Tag t3 = new Tag(3); // At end
}
```

[3] In contrast, C++ has the *constructor initializer list* that causes initialization to occur before entering the constructor body, and is enforced for objects. See *Thinking in C++, 2nd edition* (available on this book's CD ROM and at *www.BruceEckel.com*).

```
public class OrderOfInitialization {
  public static void Main() {
    Card t = new Card();
    t.F(); // Shows that construction is done
  }
} ///:~
```

In **Card**, the definitions of the **Tag** objects are intentionally scattered about to prove that they'll all get initialized before the constructor is entered or anything else can happen. In addition, **t3** is reinitialized inside the constructor. The output is:

```
Tag(1)
Tag(2)
Tag(3)
Card()
Tag(33)
f()
```

Thus, the **t3** reference gets initialized twice, once before and once during the constructor call. (The first object is dropped, so it can be garbage-collected later.) This might not seem efficient at first, but it guarantees proper initialization—what would happen if an overloaded constructor were defined that did *not* initialize **t3** and there wasn't a "default" initialization for **t3** in its definition?

Static data initialization

When the data is **static** the same thing happens; if it's a primitive and you don't initialize it, it gets the standard primitive initial values. If it's a reference to an object, it's **null** unless you create a new object and attach your reference to it.

If you want to place initialization at the point of definition, it looks the same as for non-**static**s. There's only a single piece of storage for a **static**, regardless of how many objects are created. But the question arises of when the **static** storage gets initialized. An example makes this question clear:

```
//:c04:StaticInitialization.cs
// Specifying initial values in a
// class definition.
```

```
class Bowl {
  internal Bowl(int marker) {
    System.Console.WriteLine("Bowl(" + marker + ")");
  }
  internal void F(int marker) {
    System.Console.WriteLine("F(" + marker + ")");
  }
}

class Table {
  static Bowl b1 = new Bowl(1);
  internal Table() {
    System.Console.WriteLine("Table()");
    b2.F(1);
  }
  internal void F2(int marker) {
    System.Console.WriteLine("F2(" + marker + ")");
  }
  static Bowl b2 = new Bowl(2);
}

class Cupboard {
  Bowl b3 = new Bowl(3);
  static Bowl b4 = new Bowl(4);
  internal Cupboard() {
    System.Console.WriteLine("Cupboard()");
    b4.F(2);
  }
  internal void F3(int marker) {
    System.Console.WriteLine("F3(" + marker + ")");
  }
  static Bowl b5 = new Bowl(5);
}

public class StaticInitialization {
  public static void Main() {
    System.Console.WriteLine(
      "Creating new Cupboard() in main");
    new Cupboard();
    System.Console.WriteLine(
```

```
      "Creating new Cupboard() in main");
    new Cupboard();
    t2.F2(1);
    t3.F3(1);
  }
  static Table t2 = new Table();
  static Cupboard t3 = new Cupboard();
} ///:~
```

Bowl allows you to view the creation of a class, and **Table** and **Cupboard** create **static** members of **Bowl** scattered through their class definitions. Note that **Cupboard** creates a non-**static Bowl b3** prior to the **static** definitions. The output shows what happens:

```
Bowl(1)
Bowl(2)
Table()
f(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
f(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()
f(2)
f2(1)
f3(1)
```

The **static** initialization occurs only if it's necessary. If you don't create a **Table** object and you never refer to **Table.b1** or **Table.b2**, the **static Bowl b1** and **b2** will never be created. However, they are initialized only when the *first* **Table** object is created (or the first **static** access occurs). After that, the **static** objects are not reinitialized.

The order of initialization is **static**s first, if they haven't already been initialized by a previous object creation, and then the non-**static** objects. You can see the evidence of this in the output.

It's helpful to summarize the process of creating an object. Consider a class called **Dog**:

7. The first time an object of type **Dog** is created, *or* the first time a **static** method or **static** field of class **Dog** is accessed, the C# runtime must locate the assembly in which **Dog**'s class definition is stored.

8. As the **Dog** class is loaded (creating a **Type** object, which you'll learn about later), all of its **static** initializers are run. Thus, **static** initialization takes place only once, as the **Type** object is loaded for the first time.

9. When you create a **new Dog( )**, the construction process for a **Dog** object first allocates enough storage for a **Dog** object on the heap.

10. This storage is wiped to zero, automatically setting all the primitives in that **Dog** object to their default values (zero for numbers and the equivalent for **boolean** and **char**) and the references to **null**.

11. Any initializations that occur at the point of field definition are executed.

12. Constructors are executed. As you shall see in Chapter #inheritance#, this might actually involve a fair amount of activity, especially when inheritance is involved.

Static Constructor

C# allows you to group other **static** initializations inside a special "**static** constructor." It looks like this:

```
class Spoon {
  static int i;
  static Spoon(){
    i = 47;
```

```
  }
  // . . .
```

This code, like other **static** initializations, is executed only once, the first time you make an object of that class *or* the first time you access a **static** member of that class (even if you never make an object of that class). For example:

```
//:c04:StaticConstructor.cs
// Explicit static initialization
// with static constructor

class Cup {
  internal Cup(int marker) {
    System.Console.WriteLine("Cup(" + marker + ")");
  }
  internal void F(int marker) {
    System.Console.WriteLine("f(" + marker + ")");
  }
}

class Cups {
  internal static Cup c1;
  static Cup c2;
  static Cups(){
    System.Console.WriteLine("Inside static Cups()
constructor");
    c1 = new Cup(1);
    c2 = new Cup(2);
  }
  Cups() {
    System.Console.WriteLine("Cups()");
  }
}

public class ExplicitStatic {
  public static void Main() {
    System.Console.WriteLine("Inside Main()");
    Cups.c1.F(99);  // (1)
  }
  // static Cups x = new Cups();  // (2)
```

```
  // static Cups y = new Cups();  // (2)
} ///:~
```

The **static** constructor for **Cups** run when either the access of the **static** object **c1** occurs on the line marked (1), or if line (1) is commented out and the lines marked (2) are uncommented. If both (1) and (2) are commented out, the **static** constructor for **Cups** never occurs. Also, it doesn't matter if one or both of the lines marked (2) are uncommented; the static initialization only occurs once.

Array initialization

Initializing arrays in C is error-prone and tedious. C++ uses *aggregate initialization* to make it much safer[4]. C# has no "aggregates" like C++, since everything is an object in Java. It does have arrays, and these are supported with array initialization.

An array is simply a sequence of either objects or primitives, all the same type and packaged together under one identifier name. Arrays are defined and used with the square-brackets *indexing operator* **[ ]**. To define an array you simply follow your type name with empty square brackets:

```
int[] a1;
```

You can also put the square brackets after the identifier to produce exactly the same meaning:

```
int a1[];
```

This conforms to expectations from C and C++ programmers. The former style, however, is probably a more sensible syntax, since it says that the type is "an **int** array." That style will be used in this book.

The compiler doesn't allow you to tell it how big the array is. This brings us back to that issue of "references." All that you have at this point is a reference to an array, and there's been no space allocated for the array. To create storage for the array you must write an initialization expression.

---

[4] See *Thinking in C++, 2nd edition* for a complete description of C++ aggregate initialization.

For arrays, initialization can appear anywhere in your code, but you can also use a special kind of initialization expression that must occur at the point where the array is created. This special initialization is a set of values surrounded by curly braces. The storage allocation (the equivalent of using **new**) is taken care of by the compiler in this case. For example:

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

So why would you ever define an array reference without an array?

```
int[] a2;
```

Well, it's possible to assign one array to another in C#, so you can say:

```
a2 = a1;
```

What you're really doing is copying a reference, as demonstrated here:

```
//:c04:Arrays.cs
// Arrays of primitives.

public class Arrays {
  public static void Main() {
    int[] a1 = { 1, 2, 3, 4, 5 };
    int[] a2;
    a2 = a1;
    for(int i = 0; i < a2.Length; i++)
      a2[i]++;
    for(int i = 0; i < a1.Length; i++)
      System.Console.WriteLine(
        "a1[" + i + "] = " + a1[i]);
  }
} ///:~
```

You can see that **a1** is given an initialization value while **a2** is not; **a2** is assigned later—in this case, to another array.

There's something new here: all arrays have a property (whether they're arrays of objects or arrays of primitives) that you can query—but not change—to tell you how many elements there are in the array. This member is **Length**. Since arrays in C#, as in Java and C, start counting from element zero, the largest element you can index is **Length - 1**. If you

go out of bounds, C and C++ quietly accept this and allow you to stomp all over your memory, which is the source of many infamous bugs. However, C# protects you against such problems by causing a run-time error (an *exception*, the subject of Chapter 10) if you step out of bounds. Of course, checking every array access costs time and code, which means that array accesses might be a source of inefficiency in your program if they occur at a critical juncture. Sometimes the JIT can "precheck" to ensure that all index values in a loop will never exceed the array bounds, but in general, array access pays a small performance price. By explicitly moving to "unsafe" code (discussed in #unsafe code#), bounds checking can be turned off.

What if you don't know how many elements you're going to need in your array while you're writing the program? You simply use **new** to create the elements in the array. Here, **new** works even though it's creating an array of primitives (**new** won't create a nonarray primitive):

```
//:c04:ArrayNew.cs
// Creating arrays with new.
using System;

public class ArrayNew {
  static Random rand = new Random();

  public static void Main() {
    int[] a;
    a = new int[rand.Next(20) + 1];
    System.Console.WriteLine(
      "length of a = " + a.Length);
    for(int i = 0; i < a.Length; i++)
      System.Console.WriteLine(
        "a[" + i + "] = " + a[i]);
  }
}
///:~
```

Since the size of the array is chosen at random, it's clear that array creation is actually happening at run-time. In addition, you'll see from the output of this program that array elements of primitive types are automatically initialized to "empty" values. (For numerics and **char**, this is zero, and for **boolean**, it's **false**.)

If you're dealing with an array of nonprimitive objects, you must always use **new**. Here, the reference issue comes up again because what you create is an array of references. Consider the wrapper type **IntHolder,** which is a class and not a primitive:

```
//:c04:ArrayClassObj.cs
// Creating an array of nonprimitive objects.
using System;

class IntHolder{
    int i;
    internal IntHolder(int i){
        this.i = i;
    }

    public override string ToString(){
        return i.ToString();
    }
}

public class ArrayClassObj {
  static Random rand = new Random();

  public static void Main() {
    IntHolder[] a = new IntHolder[rand.Next(20) + 1];
    System.Console.WriteLine(
      "length of a = " + a.Length);
    for(int i = 0; i < a.Length; i++) {
      a[i] = new IntHolder(rand.Next(500));
      System.Console.WriteLine(
        "a[" + i + "] = " + a[i]);
    }
  }
} ///:~
```

Here, even after **new** is called to create the array:

```
IntHolder[] a = new IntHolder[rand.Next(20) + 1];
```

it's only an array of references, and not until the reference itself is initialized by creating a new **Integer** object is the initialization complete:

```
a[i] = new IntHolder(rand.Next(500));
```

If you forget to create the object, however, you'll get an exception at run-time when you try to read the empty array location.

It's also possible to initialize arrays of objects using the curly-brace-enclosed list. There are two forms:

```
//:c04:ArrayInit.cs
// Array initialization.

class IntHolder{
    int i;
    internal IntHolder(int i){
        this.i = i;
    }

    public override string ToString(){
        return i.ToString();
    }
}


public class ArrayInit {
  public static void Main() {
    IntHolder[] a = {
      new IntHolder(1),
      new IntHolder(2),
      new IntHolder(3),
    };

    IntHolder[] b = new IntHolder[] {
      new IntHolder(1),
      new IntHolder(2),
      new IntHolder(3),
    };
  }
} ///:~
```

This is useful at times, but it's more limited since the size of the array is determined at compile-time. The final comma in the list of initializers is optional. (This feature makes for easier maintenance of long lists.)

Multidimensional arrays

C# allows you to easily create multidimensional arrays:

```
//:c04:MultiDimArray.cs
// Creating multidimensional arrays.
using System;

class IntHolder {
    int i;
    internal IntHolder(int i) {
        this.i = i;
    }

    public override string ToString() {
        return i.ToString();
    }
}

public class MultiDimArray {
    static Random rand = new Random();

    static void Prt(string s) {
        System.Console.WriteLine(s);
    }
    public static void Main() {
        int[,] a1 = {
            { 1, 2, 3,},
            { 4, 5, 6,},
        };
        Prt("a1.Length = " + a1.Length);
        Prt(" == " + a1.GetLength(0) + " * " +
a1.GetLength(1));
        for (int i = 0; i < a1.GetLength(0); i++)
            for (int j = 0; j < a1.GetLength(1); j++)
                Prt("a1[" + i + "," + j +
                    "] = " + a1[i, j]);
```

```
// 3-D rectangular array:
int[,,] a2 = new int[2, 2, 4];
for (int i = 0; i < a2.GetLength(0); i++)
    for (int j = 0; j < a2.GetLength(1); j++)
        for (int k = 0; k < a2.GetLength(2);
            k++)
            Prt("a2[" + i + "," +
                j + "," + k +
                "] = " + a2[i,j,k]);
// Jagged array with varied-Length vectors:
int[][][] a3 = new int[rand.Next(7) + 1][][];
for (int i = 0; i < a3.Length; i++) {
    a3[i] = new int[rand.Next(5) + 1][];
    for (int j = 0; j < a3[i].Length; j++)
        a3[i][j] = new int[rand.Next(5) + 1];
}
for (int i = 0; i < a3.Length; i++)
    for (int j = 0; j < a3[i].Length; j++)
        for (int k = 0; k < a3[i][j].Length;
            k++)
            Prt("a3[" + i + "][" +
                j + "][" + k +
                "] = " + a3[i][j][k]);
// Array of nonprimitive objects:
IntHolder[,] a4 = {
    { new IntHolder(1), new IntHolder(2)},
    { new IntHolder(3), new IntHolder(4)},
    { new IntHolder(5), new IntHolder(6)},
};
for (int i = 0; i < a4.GetLength(0); i++)
    for (int j = 0; j < a4.GetLength(1); j++)
        Prt("a4[" + i + "," + j +
            "] = " + a4[i,j]);
IntHolder[][] a5;
a5 = new IntHolder[3][];
for (int i = 0; i < a5.Length; i++) {
    a5[i] = new IntHolder[3];
    for (int j = 0; j < a5[i].Length; j++){
        a5[i][j] = new IntHolder(i*j);
    }
}
```

```
        for (int i = 0; i < a5.GetLength(0); i++){
            for (int j = 0; j < a5[i].Length; j++){
                Prt("a5[" + i + "][" + j +
                    "] = " + a5[i][j]);
            }
        }
    }
} ///:~
//@todo: Fix MultiDimArray to deal with rectangular
and jagged arrays. Discuss performance ramifications
```

The code used for printing uses **Length** so that it doesn't depend on fixed array sizes.

The first example shows a multidimensional array of primitives. You delimit each vector in the array with curly braces:

```
    int[][] a1 = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
```

Each set of square brackets moves you into the next level of the array.

The second example shows a three-dimensional array allocated with **new**. Here, the whole array is allocated at once:

```
int[][][] a2 = new int[2][2][4];
```

But the third example shows that each vector in the arrays that make up the matrix can be of any length:

```
    int[][][] a3 = new int[pRand(7)][][];
    for(int i = 0; i < a3.length; i++) {
      a3[i] = new int[pRand(5)][];
      for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[pRand(5)];
    }
```

The first **new** creates an array with a random-length first element and the rest undetermined. The second **new** inside the **for** loop fills out the elements but leaves the third index undetermined until you hit the third **new**.

You will see from the output that array values are automatically initialized to zero if you don't give them an explicit initialization value.

You can deal with arrays of nonprimitive objects in a similar fashion, which is shown in the fourth example, demonstrating the ability to collect many **new** expressions with curly braces:

```
Integer[][] a4 = {
  { new Integer(1), new Integer(2)},
  { new Integer(3), new Integer(4)},
  { new Integer(5), new Integer(6)},
};
```

The fifth example shows how an array of nonprimitive objects can be built up piece by piece:

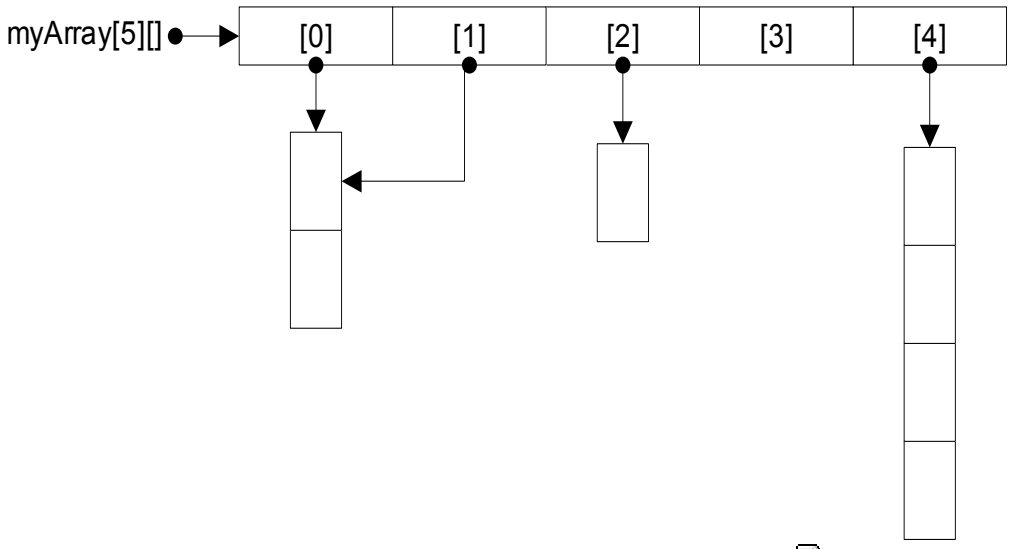
```
Integer[][] a5;
a5 = new Integer[3][];
for(int i = 0; i < a5.length; i++) {
  a5[i] = new Integer[3];
  for(int j = 0; j < a5[i].length; j++)
    a5[i][j] = new Integer(i*j);
}
```

The **i*j** is just to put an interesting value into the **Integer**.

Sidebar/Appendix: What a difference a rectangle makes

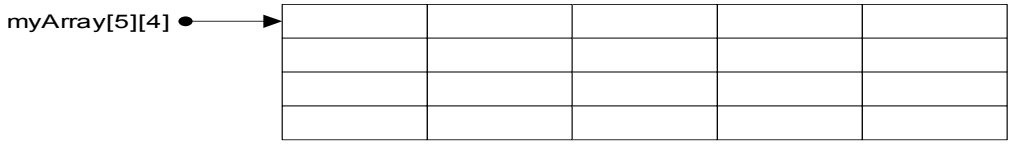
The addition of "rectangular" arrays to C# is one of a few different language features that have the potential to make C# a great language for numerically intensive computing. With "jagged" arrays (arrays of the form Object[][]), it's impossible for an optimizer to make assumptions about memory allocation. A jagged array may have multiple rows pointing to the same base array, unallocated rows, and cross-references.

```
double[5][] myArray = new double[5][];
myArray[0] = new double[2];
myArray[1] = myArray[0];
myArray[2] = new double[1];
myArray[4] = new double[4];
```



A rectangular array, on the other hand, is a contiguous block:

```
double myArray[5,4] = new double[5,4];
```



Several optimizing techniques are harder to do with jagged arrays than with rectangular. When researchers at IBM added rectangular arrays to Java, they speeded up some numerical benchmarks by factors close to 50! ("The Ninja Project," Moreira et al., CACM, Oct 2001) So far, the C# optimizer doesn't seem to be taking advantage of such possibilities, although it does run somewhat faster than Java on Cholesky multiplication, one of the benchmarks used by the researchers, ported here to C#:

```
//:c04:Cholesky.cs
/* Demonstration program showing use of the Array
package             */
/*
*/
/* CHOLESKY: This program computes the Cholesky factor
U of a         */
```

```
/* symmetric positive definite matrix A such that A =
U'U where       */
/* U' is the transpose of U. U is an upper triangular
matrix.         */
using System;

public class Cholesky {

  public static void Main() {
    int n = 1000;                    // problem size
    double[,] A = new double[n,n]; // n x n 2-
dimensional array
    double ops;
    double mflops;

    /*
     * Compute Cholesky factor using an optimized
code.
     */
    initialize(A,n);
    //Accurate w/i 10ms on NT/2K/XP, w/i 55ms on
Win95/98
    DateTime t1 = DateTime.Now;
    unroll(A,n);
    DateTime t2 = DateTime.Now;
    TimeSpan time = (t2 - t1);
    ops = ((((1.0 * n) * n) * n) / 3.0);
    mflops = ((1.0e-6 * ops) / time.TotalSeconds);
    System.Console.Write("Optimized code, Time:
"+time+" Mflops:"+mflops+" ");
    validate(A,n);

    /*
     * Compute Cholesky factor using a straightforward
code.
     */
    initialize(A,n);
    t1 = DateTime.Now;
    plain(A,n);
    t2 = DateTime.Now;
    time = (t2 - t1);
```

```
    ops = ((((1.0 * n) * n) * n) / 3.0);
    mflops = ((1.0e-6 * ops) / time.TotalSeconds);
    System.Console.Write("Simple code, Time: "+time+"
Mflops:"+mflops+" ");
    validate(A,n);
  }

  public static void initialize(double[,] A,int n) {
    int i;
    int j;
    for( j = 0; j <= (n - 1); j++) {
      for( i = 0; i <= (n - 1); i++) {
        A[j, i] = 0;
      };
    };
    for( j = 0; j <= (n - 1); j++) {
      for( i = j; i <= (n - 1); i++) {
        A[i, j] = j + 1;
      };
    };
  }

  public static void validate(double[,] A,int n) {
    int i;
    int j;
    int result;
    double eps;
    result = 1;
    eps = 1.0e-10;
    for( i = 0; i <= (n - 1); i++) {
      for( j = i; j <= (n - 1); j++) {
        if (((Math.Abs((A[j, i] - 1.0))) > eps)) {
            System.Console.WriteLine(Math.Abs((A[j, i]
- 1.0)));
          result = 0;
        };
      };
    };
    if (result == 1) {
      System.Console.WriteLine("Result is correct!");
    } else {
```

```
      System.Console.WriteLine("Result is
incorrect!");
    }
  }

  public static void plain(double[,] A,int n) {
    int i;
    int j;
    int k;
    double s;
    double d;
    for( j = 0; j <= (n - 1); j++) {
      d = 0.0;
      for( k = 0; k <= (j - 1); k++) {
        s = 0.0;
        for( i = 0; i <= (k - 1); i++) {
          s = (s + (A[k,i] * A[j,i]));
        };
        s = ((A[j,k] - s) / A[k,k]);
        A[j,k] = s;
        d = (d + (s * s));
      };
      d = (A[j,j] - d);
      A[j,j] = Math.Sqrt(d);
    };
  }

  public static void unroll(double[,] A,int n) {
    int i;
    int j;
    int k;
    double s;
    double d;
    double s0;
    double s1;
    double s2;
    double s3;
    double aij;
    for( j = 0; j <= (n - 1); j++) {
      d = 0.0;
      for( k = 0; k <= (j - 4); k += 4) {
```

```
        s0 = 0.0;
        s1 = 0.0;
        s2 = 0.0;
        s3 = 0.0;
        for( i = 0; i <= (k - 1); i++) {
          aij = A[j,i];
          s0 = (s0 + (A[(k + 0),i] * aij));
          s1 = (s1 + (A[(k + 1),i] * aij));
          s2 = (s2 + (A[(k + 2),i] * aij));
          s3 = (s3 + (A[(k + 3),i] * aij));
        };
        s0 = ((A[j,(k + 0)] - s0) / A[(k + 0),(k +
0)]);
        A[j,(k + 0)] = s0;
        s1 = (s1 + (A[(k + 1),k] * A[j,k]));
        s1 = ((A[j,(k + 1)] - s1) / A[(k + 1),(k +
1)]);
        A[j, k + 1] = s1;
        s2 = ((s2 + (A[(k + 2),k] * A[j,k])) + (A[(k +
2),(k + 1)] * A[j,(k + 1)]));
        s2 = ((A[j,(k + 2)] - s2) / A[(k + 2),(k +
2)]);
        A[j,(k + 2)] = s2;
        s3 = (((s3 + (A[(k + 3),k] * A[j,k])) + (A[(k
+ 3),(k + 1)] * A[j,(k + 1)])) + (A[(k + 3),(k + 2)] *
A[j,(k + 2)]));
        s3 = ((A[j,(k + 3)] - s3) / A[(k + 3),(k +
3)]);
        A[j,(k + 3)] = s3;
        d = ((((d + (s0 * s0)) + (s1 * s1)) + (s2 *
s2)) + (s3 * s3));
      };
      for( k = ((j / 4) * 4); k <= (j - 1); k++) {
        s = 0.0;
        for( i = 0; i <= (k - 1); i++) {
          s = (s + (A[k,i] * A[j,i]));
        };
        s = ((A[j,k] - s) / A[k,k]);
        A[j,k] = s;
        d = (d + (s * s));
      };
```

```
      d = (A[j,j] - d);
      A[j,j] = (Math.Sqrt(d));
    };
  }
}
///:~
```

For comparison, here's an equivalent program in Java, using jagged arrays:

```
//:c04:Cholesky.java
/* Demonstration program showing use of the Array
package             */
/*
*/
/* CHOLESKY: This program computes the Cholesky factor
U of a          */
/* symmetric positive definite matrix A such that A =
U'U where       */
/* U' is the transpose of U. U is an upper triangular
matrix.         */
import java.math.*;

public class Cholesky_native_array {

  public static void main(String[] args) {
    int n = 1000;                    // problem size
    double[][] A = new double[n][n]; // n x n 2-
dimensional array
    double ops;
    double mflops;

    /*
     * Compute Cholesky factor using an optimized
code.
     */
    initialize(A,n);
    //Accurate w/i 10ms on NT/2K/XP, w/i 55ms on
Win95/98
    double t1 = (1e-3*System.currentTimeMillis());
    unroll(A,n);
```

```
    double t2 = (1e-3*System.currentTimeMillis());
    double time = (t2 - t1);
    ops = ((((1.0 * n) * n) * n) / 3.0);
    mflops = ((1.0e-6 * ops) / time);
    System.out.print("Optimized code, Time: "+time+"
Mflops:"+mflops+" ");
    validate(A,n);

    /*
     * Compute Cholesky factor using a straightforward
code.
     */
    initialize(A,n);
    t1 = (1e-3*System.currentTimeMillis());
    plain(A,n);
    t2 = (1e-3*System.currentTimeMillis());
    time = (t2 - t1);
    ops = ((((1.0 * n) * n) * n) / 3.0);
    mflops = ((1.0e-6 * ops) / time);
    System.out.print("Simple code, Time: "+time+"
Mflops:"+mflops+" ");
    validate(A,n);
  }

  public static void initialize(double[][] A,int n) {
    int i;
    int j;
    for( j = 0; j <= (n - 1); j++) {
      for( i = 0; i <= (n - 1); i++) {
        A[j][i] = 0;
      };
    };
    for( j = 0; j <= (n - 1); j++) {
      for( i = j; i <= (n - 1); i++) {
        A[i][j] = j + 1;
      };
    };
  }

  public static void validate(double[][] A,int n) {
    int i;
```

```
    int j;
    int result;
    double eps;
    result = 1;
    eps = 1.0e-10;
    for( i = 0; i <= (n - 1); i++) {
      for( j = i; j <= (n - 1); j++) {
        if (((Math.abs((A[j][i] - 1.0))) > eps)) {
          result = 0;
        };
      };
    };
    if (result == 1) {
      System.out.println("Result is correct!");
    } else {
      System.out.println("Result is incorrect!");
    }
  }

  public static void plain(double[][] A,int n) {
    int i;
    int j;
    int k;
    double s;
    double d;
    for( j = 0; j <= (n - 1); j++) {
      d = 0.0;
      for( k = 0; k <= (j - 1); k++) {
        s = 0.0;
        for( i = 0; i <= (k - 1); i++) {
          s = (s + (A[k][i] * A[j][i]));
        };
        s = ((A[j][k] - s) / A[k][k]);
        A[j][k] = s;
        d = (d + (s * s));
      };
      d = (A[j][j] - d);
      A[j][j] = Math.sqrt(d);
    };
  }
```

```
  public static void unroll(double[][] A,int n) {
    int i;
    int j;
    int k;
    double s;
    double d;
    double s0;
    double s1;
    double s2;
    double s3;
    double aij;
    for( j = 0; j <= (n - 1); j++) {
      d = 0.0;
      for( k = 0; k <= (j - 4); k += 4) {
        s0 = 0.0;
        s1 = 0.0;
        s2 = 0.0;
        s3 = 0.0;
        for( i = 0; i <= (k - 1); i++) {
          aij = A[j][i];
          s0 = (s0 + (A[(k + 0)][i] * aij));
          s1 = (s1 + (A[(k + 1)][i] * aij));
          s2 = (s2 + (A[(k + 2)][i] * aij));
          s3 = (s3 + (A[(k + 3)][i] * aij));
        };
        s0 = ((A[j][(k + 0)] - s0) / A[(k + 0)][(k +
0)]);
        A[j][(k + 0)] = s0;
        s1 = (s1 + (A[(k + 1)][k] * A[j][k]));
        s1 = ((A[j][(k + 1)] - s1) / A[(k + 1)][(k +
1)]);
        A[j][k + 1] = s1;
        s2 = ((s2 + (A[(k + 2)][k] * A[j][k])) + (A[(k
+ 2)][(k + 1)] * A[j][(k + 1)]));
        s2 = ((A[j][(k + 2)] - s2) / A[(k + 2)][(k +
2)]);
        A[j][(k + 2)] = s2;
        s3 = (((s3 + (A[(k + 3)][k] * A[j][k])) +
(A[(k + 3)][(k + 1)] * A[j][(k + 1)])) + (A[(k +
3)][(k + 2)] * A[j][(k + 2)]));
```

```
        s3 = ((A[j][(k + 3)] - s3) / A[(k + 3)][(k +
3)]);
        A[j][(k + 3)] = s3;
        d = ((((d + (s0 * s0)) + (s1 * s1)) + (s2 *
s2)) + (s3 * s3));
      };
      for( k = ((j / 4) * 4); k <= (j - 1); k++) {
        s = 0.0;
        for( i = 0; i <= (k - 1); i++) {
          s = (s + (A[k][i] * A[j][i]));
        };
        s = ((A[j][k] - s) / A[k][k]);
        A[j][k] = s;
        d = (d + (s * s));
      };
      d = (A[j][j] - d);
      A[j][j] = (Math.sqrt(d));
    };
  }
}///:~
```

When the two programs are run on my P3-500, these are the results:

Cholesky.exe:

java Cholesky_native_array:

Summary

This seemingly elaborate mechanism for initialization, the constructor, should give you a strong hint about the critical importance placed on initialization in the language. As Stroustrup was designing C++, one of the first observations he made about productivity in C was that improper initialization of variables causes a significant portion of programming problems. These kinds of bugs are hard to find, and similar issues apply to improper cleanup. Because constructors allow you to *guarantee* proper initialization and cleanup (the compiler will not allow an object to be created without the proper constructor calls), you get complete control and safety.

In C++, destruction is quite important because objects created with **new** must be explicitly destroyed. In C#, the garbage collector automatically releases the memory for all objects, so the equivalent cleanup method in C# isn't necessary much of the time. In cases where you don't need destructor-like behavior, C#'s garbage collector greatly simplifies programming, and adds much-needed safety in managing memory. However, the garbage collector does add a run-time cost, the expense of which is difficult to put into perspective because of the other performance ramifications of the IL and CRL approach to binary files.

Because of the guarantee that all objects will be constructed, there's actually more to the constructor than what is shown here. In particular, when you create new classes using either *composition* or *inheritance* the guarantee of construction also holds, and some additional syntax is necessary to support this. You'll learn about composition, inheritance, and how they affect constructors in future chapters.

Exercises

6: Coupling and Cohesion

Why do people worry about software “architecture”? What makes software hard to understand? How does controlling scope and creating abstract interfaces make even the most complex problem tractable? What are the types of coupling and their relative costs? What are the types of cohesion and their relative benefits? How do objects address these issues? How does good design emerge from four simple rules about code structure? How C# organizes files and types. Why namespaces are a good approach to scope beyond the object-level. Creating your own namespaces. The concept of visibility. The fifth rule of code structure. Public, internal, and private access modifiers (protected and protected internal are deferred to the next chapter).

Data encapsulation, inheritance, and polymorphism are the cornerstone capabilities of object orientation, but their use does not automatically create good or even passable design. Good software design arises from dividing a problem into coherent parts and tackling those parts independently, in a manner that's easy to test and change. Object orientation facilitates just such partitioning of concerns, but requires you to understand the forces that make some software designs better than others. These forces, coupling and cohesion, are universal to all software systems and provide a basis for comparing software architectures and designs.

When we speak of "architecture" and "design" in software systems, we can be speaking of many different things – the physical structure of the network, the set of operating system and server providers chosen, the graphic design of the entire client-facing Website, the user-interface design of applications we write, or the internal structure of the programs being written. All of these factors are important and it's a shame that we have not yet developed a common vocabulary for giving each of them their just attention. For the purposes of this chapter, we are solely concerned with the internal structure of programs – decisions that will be made and embodied in the C# source code you write.

Software As Architecture vs. Software Architecture

@todo: Move this somewhere else

As often as not, the lead programmer in a team has the title of Software Architect. The popularity of this title comes from the popular view that the challenges of building a software system are similar to the challenges of building a skyscraper or bridge.

The view of software *as* architecture stems from the undeniable truth that software often fails. Most of those who have studied larger software projects (projects in excess of 5,000 function points, which translates to probably 175,000 lines of C#) have concluded that aspects other than code construction are the most important drivers of success or failure. What these people see in large projects, time and again, are failures relating to requirements and failures relating to integration: teams build the wrong thing and then they can't get the pieces they build to fit together. Based on these observations, the challenges of software seemed to parallel the challenges of architecture – the craftsmanship of the individual worker is all well and good, but success comes from careful planning and coordination.

Every few years, the pendulum of popular opinion swings from this view towards a view that emphasizes the individual contributions of talented programmers. A few

years later, the pendulum predictably swings the other way. This view of software holds that, like it or not, code defines the software's behavior and that diagrams and specifications do not generally capture the real issues at hand when designing programs. Proponents of this view argue that the software *as* architecture view is primarily pushed by consultants and vendors selling expensive tools that generate a lot of paper but little product.

Until recently, the escalation of defect costs over time was the trump card for the software *as* architecture advocates. In 1987, Barry Boehm published a paper in which he established that a defect corrected for $1 in the requirements phase could cost $100 to fix once deployed[1]. With those economics, big up-front efforts were obviously worthwhile. It's doubtful that such numbers have much meaning today; Boehm himself concluded in 2001 that for "smaller, noncritical software systems" the cost escalation was "more like 5:1 than 100:1." The explosively popular set of practices called Extreme Programming is based on "the technical premise [that] the cost of change [can rise] much more slowly, eventually reaching an asymptote."[2]

So should you view software development as an undertaking akin to building a bridge or as one akin to growing a garden? Well, there are a few things we can say for sure about software development: Programmers always overestimate their long-term productivity. Most large software projects still cost more than expected. Most projects still take longer to finish than expected. Many are cancelled prior to completion and many, once deployed, fail to achieve the business goals they were intended to serve. Beyond these generally pessimistic statements, the variances between individuals, teams, companies, and industry sectors swamp the statistics. Useful software systems range from programs that are a few hundred lines long (the source code to **ping** is shorter than some of the samples in this book!) to programs in excess of a million lines of code. Teams can range from individuals to hundreds. A successful program may be a function-optimizer that, on a single run out of a thousand, produces a meaningful result or a life support system that, literally, never fails (yes, provably correct software is possible). So when people start spouting statistics or pronouncements about "best practices," at best they're over-generalizing and at worst they're trying to scare you into buying whatever it is they're selling[3].

We do know that there's at least one major flaw in the software as architecture metaphor, and that is that unlike a building that remains under construction until a designated "grand opening," software should be built and deployed incrementally – if not one "room at a time" then at least "one floor at a time." This is, of course, easier

---

[1] B. Boehm and V. Basili, "Software Defect Reduction Top 10 List," IEEE Computer, IEEE Computer Society, Vol. 34, No. 1, January 2001, pp. 135-137.

[2] Extreme Programming Explained, pg. 23

[3] Be especially dubious if you hear "70% failure rate." This, like the $300B that Y2K failures were going to cost, is a number that originated in a single profit-motivated study but which has been repeated so many times that it's taken on a life of its own.

in some situations than in others, but one of the great truisms of software development management is that from the first weeks of the project, you should be able to at least potentially ship your latest build.

What Is Software Architecture

Every non-trivial program should have an overall ordering principle. While that's vague enough to allow for a lot of different interpretations of what does and does not constitute an architecture, you don't gain much by viewing .NET as your systems' architecture. Rather, you should consider the way that data flows in and out of your system and the ways in which it is transformed. At such a level of abstraction, there are fewer variations in structure than you might guess.

Most programs are going to have at least a few subsystems, and each may have its own architecture. For instance, almost all systems require some kind of user interface, even if it's just for administration, and UI subsystems are typically architected to follow either the Model-View-Controller pattern or the Presentation-Abstraction-Control pattern (both of which are thoroughly explained in Chapter #Windows forms#). However, neither of these architectures speaks to how the program should structure the creation of business value (what goes on in the "Model" or "Abstraction" parts of MVC and PAC respectively).

Simulation Architectures: Always Taught, Rarely Used

Many people are taught that object oriented systems should be structured to conform to objects in the problem domain, an architectural pattern we could call Simulation. There is certainly historical precedent for this: object oriented programming was in fact invented in response to the needs of simulation programming. Unfortunately, outside of video games and certain types of modeling, Simulation is not generally a useful architecture. Why? Well, because the business interests of most companies are usually focused on transactions driven by non-deterministic forces (i.e., people) and it just doesn't do much for the bottom line to try to recreate those forces inside the computer system. Amazon's ordering system probably has a **Book** class, and maybe an **Author** class, but I'll wager it doesn't have an **Oprah** class, no matter how much the talk-show host's recommendations influence purchasers.

Simulation architectures tend to exemplify the concept of cohesion, though. Cohesion is the amount of consistency and completion in a class or namespace, the amount by which a class or namespace "hangs together." In their proper domain, simulations tend to have great cohesion: when you've got a **Plant** object and an **Animal** object, it's pretty easy to figure out where to put your **photosynthesis()** method. Similarly, with classes derived from real world nouns, one doesn't often

have “coincidental cohesion” where a class contains multiple, completely unrelated interfaces.

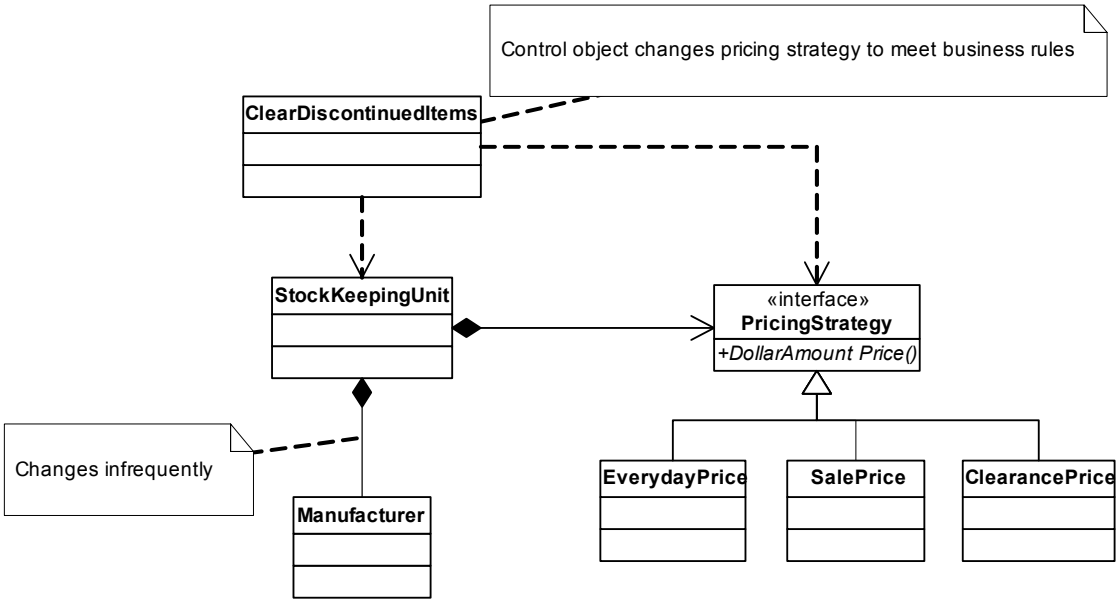
Client/Server and n-Tier Architectures

When corporate networks first became ubiquitous in the early 1990s, there was a great deal of excitement about the possibility of exploiting the desktop computers’ CPU power and display characteristics, while maintaining data integrity and security on more controllable servers. This client/server architecture quickly fell out of favor as client software turned out to be costly to develop and difficult to keep in sync with changing business rules. Client/Server was replaced with 3-Tier and n-Tier Architectures, which divide the world into 3 concerns: a presentation layer at the client, one or more business rules layers that “run anywhere” (but usually on a server), and a persistence or database layer that definitely runs on a server, but perhaps not the same machines that run the business rules layers.

Originally popular for corporate development, the dot-com explosion entrenched the n-Tier Architecture as the most common system architecture in development today. Web Services, which we’ll discuss in more detail in Chapter #Web Services#, are n-Tier Architectures where the communication between layers is done with XML over Web protocols such as SOAP, HTTP, and SMTP.

In this architecture, all the interesting stuff happens in the business-rule tiers, which are typically architected in a use-case-centric manner. “Control” classes represent the creation of business value, which is usually defined in terms of a use-case, an atomic use of the system that delivers value to the customer. These Control objects interact with objects that represent enduring business themes (such as Sales, Service, and Loyalty), financial transactions (such as Invoices, Accounts Receivable and Payable, and Taxes), regulatory constraints, and business strategies. Classes that map into nouns in the real world are few and far between.

The theme of such business-tier architectures is “isolating the vectors of change.” The goal is to create classes based not on a mapping to real-world nouns but to patterns of behavior or strategy that are likely to change over time. For instance, a retail business is likely to have some concept of a Stock Keeping Unit – a product in a particular configuration with a certain manufacturer, description, and so forth – but may have many different strategies for pricing. In a typical business-tier architecture, this might lead to the creation of classes such as **StockKeepingUnit**, **Manufacturer** , and an interface for **PricingStrategy** implemented by **EverydayPrice, SalePrice,** or **ClearancePrice**. The “vector of change” in this case is the **PricingStrategy**, which is manipulated by a control object; for instance, an object which lowers prices when a **StockKeepingUnit** is discontinued:



This design fragment is typical of the types of structures one sees in business-tier architectures: use-cases are reified (a fancy word for "instantiated as objects"), there is extensive use of the Strategy pattern, and there are lots of classes which do not map to tangible objects in the real world.

The business rule "When a product is discontinued by its manufacturer, place it on sale until we have less than a dozen in inventory, then price it for clearance," might look like this in code:

```
class ClearDiscontinuedItems{
   static readonly int MIN_BEFORE_CLEARANCE = 12;

   Discontinued(StockKeepingUnit[] line){
      foreach(StockKeepingUnit sku in line){
         int count = storeInventory.Count(line);
         if(count < MIN_BEFORE_CLEARANCE){
            sku.PricingStrategy = new ClearancePrice();
         }else{
            sku.PricingStrategy = new SalePrice();
         }
      }
   }
}
```

In a Simulation architecture, a **StockKeepingUnit** would price itself when discontinued:

```
class StockKeepingUnit{
    //...
    static int numberInStock;
    static readonly int MIN_BEFORE_CLEARANCE = 12;
```

```
    Discontinue(){
       if(numberInStock < MIN_BEFORE_CLEARANCE){
          this.PricingStrategy = new ClearancePrice();
       }else{
          this.PricingStrategy = new SalePrice();
       }
    }
}
```

As can be seen in this design fragment, business-tier architectures tend to look less cohesive than simulation architectures for small problems. In real-world applications, though, there are typically dozens or hundreds of use-cases, each of which may have several business rules associated with it. Business rules tend to involve a lot of different factors and yet are the natural unit of thought for the domain experts, so business-tier architectures tend to result in clearer lines of communication between development teams and domain experts. "We need to ask a question about the 'price discontinued items' business rules" is much clearer to domain experts than "We need to ask a question about how an SKU prices itself when it's discontinued."

Layered Architectures

Once upon a time, computers were primarily used to automate repetitive calculations: tide charts, gunnery tables, census statistics. Nowadays, CPU-intensive math is one of the least frequent tasks asked of a computer. I imagine there are many younger people who don't know that, essentially, computers are nothing but binary calculators. This is because layer after layer of increasing abstraction has been placed between the end-user and the underlying hardware. I'm not sure this is entirely beneficial when it comes to programming, but I certainly don't want to go back to the days when the big trick in graphics programming was doing your work when the screen was doing a vertical retrace. Layers of abstraction are among the most powerful architectures over the long term, removing entire areas from the concerns of later programmers.

Layered architectures are commonly used when implementing protocols, since protocols themselves have a tendency to be layered on top of other protocols. Layers are also ubiquitous when working with device drivers or embedded systems (not likely scenarios for C# today, but who knows what tomorrow holds?).

The three places where I can anticipate a C# team looking towards layered architectures are in the areas of object-relational mapping, concurrent libraries, and network communications. The first two are areas where the .NET framework does not provide sufficient abstraction and the last is a fast-moving area where C# has the potential to really shine.

Layered architectures always seem to be difficult to get right. In a layered architecture, all classes in a layer should share the same abstraction layer, and each layer should be able to perform all its functions by only relying on its own classes and

those of the layer immediately "below" it. Pulling this off requires a more complete understanding of the problem domain than is typically available in the early stages of system development, when architecture is decided. So usually with a layered architecture, you face two possibilities: a lot of upfront effort or being very conscientious about moving objects and refactoring based on abstraction level.

The great benefits of a layered architecture come down the line when an understanding of the working of the lower levels is rarely or never needed. This is the situation today with, say, graphics, where no one but game programmers need concern themselves with the complex details of screen modes, pixel aspects, and display pages (and with modern game engines built on DirectX and the prevalence of graphics accelerator cards, apparently even game programmers are moving away from this level of detail).

Problem-Solving Architectures

It's increasingly rare to develop a system that solves a problem that is otherwise intractable. This is too bad, because there are few thrills greater than developing a tool and watching that tool accomplish something beyond your capabilities. Such systems often require the development of specialized problem-solving architectures -- blackboard systems, function optimizers, fuzzy logic engines, support-vector machines, and so forth. These problem solvers may themselves be architected in any number of ways, so in a sense you could say that this is a variation of the layered architecture, where the problem-solver is a foundation on which the inputs and meta-behavior rely. But to solve a problem using, say, a genetic optimizer or expert system requires an understanding of the characteristics of the problem-solver, not so much an understanding of the way that the problem-solver is implemented. So if you're fortunate enough to be working in such a domain, you should concentrate on understanding the core algorithms and seeing if there's a disconnect between the academic work on the problem-solver and the task at hand; many such projects fail because the problem-solver requires that the input or meta-data have a certain mathematical characteristic. As an object lesson, research on artificial neural networks languished for more than a decade because the original work used a discontinuous "activation function." Once someone realized the mathematical problem with that, a famous "proof" of their limitations fell by the wayside and neural nets became one of the hottest research topics in machine learning!

Dispatching Architectures

An amazingly large amount of value can be derived from systems that manage the flow of information without a great deal of interaction with the underlying data. These systems typically deal with the general problem of efficiently managing the flow from multiple, unknown-at-compilation-time sources of data with multiple, unknown-at-compilation-time sinks for data. "Efficiently" is sometimes measured in terms of speed, sometimes in terms of resources, and sometimes in terms of

reliability. Sometimes the connections are transient, in which case the architectures are usually called "event driven" and sometimes the connections are longer-lasting, in which case multiplexing and demultiplexing architectures are commonly used.

Dispatching solutions tend to be relatively small and can sometimes be seen as design elements, not the architectural organizing principle for an entire system or subsystem. But they often play such a crucial role in delivering value that they deserve the greatest emphasis in a system's development.

"Not Really Object-Oriented"

One of the most common criticisms that you'll hear about a piece of software is that, although it's written with an object-oriented language, "it's not really object oriented." There are two root causes for this criticism, one of which is serious and worth criticizing, and one of which betrays a lack of understanding on the part of the criticizer.

The frivolous criticism comes from those who mistakenly believe that Simulation architectures are the only valid ordering principle for object-oriented systems. Each architecture has its strengths and weaknesses and while Simulation architectures have historical ties with object orientation, they aren't the best choice in many, maybe most, real-world development efforts. There are many more architectures than have been discussed here; we've just touched on some of the more popular ones.

The serious criticism, the thing that makes too many systems "not really object oriented" is low-level design that doesn't use object-oriented features to improve the quality of the software system. There are many different types of quality: speed of execution, memory or resource efficiency, ease of integration with other code, reusability, robustness, and extensibility (to name just a few). The most important measure of software quality, though, is the extent to which your software can fulfill business goals, which usually means the extent to which the software fulfills user needs or the speed with which you can react to newly discovered user needs. Since there's nothing you can do about the former, the conclusion is "the most important thing you can do is use object-oriented features to improve the speed with which you can react to newly discovered user needs."

Design Is As Design Does

The end-user has no interest in your software's design. They care that your software helps them do their job easier. They care that your software doesn't irritate them. They care that when they talk to you about what they need from the software, you listen and respond sympathetically (and in English, not Geek). They care that your software is cheap and reliable and robust. Other than that, they don't care if it's written in assembly language, LISP, C#, or FORTRAN, and they sure as shooting don't care about your class and sequence diagrams.

So there's really *nothing* that matters about design except how easy it is to change or extend – the things that you have to do when you discover that your existing code falls short in some manner.

So how do you design for change?

First, Do No Harm

It's acceptable for you to decide not to make a change. It's acceptable for your change to turn out to be less helpful to the user than the original. It's acceptable for your change to be dead wrong, either because of miscommunication or because your fingers slipped on the keys and typed a plus instead of a minus. What is unacceptable is for you to commit a change that breaks something else in your code. And it is unacceptable for you to not know if your change breaks something else in the code. Yet such a pathological state, where the only thing that developers can say about their code is a pathetic "Well, it *shouldn't* cause anything else to change," is very common. Almost as common are situations where changes do, in fact, seem to randomly affect the behavior of the system as a whole.

It is counterintuitive, but the number one thing you can do to speed up your development schedule is to write a lot of tests. It doesn't pay off in the first days, but it pays off in just a few weeks. By the time you get several months into a project, you will be madly in love with your testing suite. Once you develop a system with a proper suite of tests, you will consider it incompetent foolishness to develop without one.

With modern languages that support reflection, it has become possible to write test infrastructure systems that discover at runtime what methods are tests (by convention, methods that begin with the string "test") and running them and reporting the results. The predominant such system is JUnit, created by Kent Beck and Erich Gamma. Several ports and refactorings of JUnit for .NET languages are available, the most popular of which is Philip Craig's NUnit. You can download NUnit from http://nunit.sourceforge.net. NUnit is a straightforward port of JUnit and does not exploit C# or .NET features such as attributes, which might be used to create new features (such as finer control of which tests are applied to a method or class).

Software systems are among the most complex structures built by humans, and unintended consequences are an inevitable part of manipulating those systems. The speed with which you can change a software system is dependent on many things, but on nothing so much as your ability to isolate a change in both .

Design Rule #1: Write Boring Code

For instance, let's say that you want to prepare your house for vacation by canceling the mail and turning on the automatic sprinkler in the garden. It seems logical that you'd do this in a method **House.VacationPrep()**:

```
//<Example>
//<Class>Vacation1.cs</Class>
namespace Vacation1{
    class House {
        Mail myMail = new Mail();
        Garden myGarden = new Garden();

        public void VacationPrep() {
            System.Console.WriteLine("Preparing the house for
vacation");
            System.Console.WriteLine("Stopping the mail");
            myMail.Active = false;
            System.Console.WriteLine("Setting the automatic
sprinker");
            myGarden.SprinklerSet = true;
            System.Console.WriteLine("House is prepared");
        }

        public static void Main() {
            House myHouse = new House();
            myHouse.VacationPrep();
        }
    }

    class Mail {
        bool active;
        public bool Active{
            get { return active;}
            set { active = value;}
        }
    }

    class Garden {
        bool sprinklerOn;
        public bool SprinklerSet{
            get { return sprinklerOn;}
            set { sprinklerOn = value;}
        }
    }
}
```

```
//</Example>
```

There's nothing wrong with this code as it stands, **House.VacationPrep()** is cohesive and not complex. But your user comes to you and says "Oh, gee, we only want to deactivate the mail if the vacation is going to be longer than 3 days." What do you do? The quickest thing to do is change House.VacationPrep() to:

```
        public void VacationPrep(int length) {
            System.Console.WriteLine("Preparing the house for
vacation");
            if(length > 3){
               System.Console.WriteLine("Stopping the mail");
               myMail.Active = false;
            }
            System.Console.WriteLine("Setting the automatic
sprinker");
            myGarden.SprinklerSet = true;
            System.Console.WriteLine("House is prepared");
        }
```

But as soon as you type that, you worry about the "magic number" 3 in the code. Magic numbers are anathema to many programmers, so much so that "DAYS_IN_ WEEK" is more common than "7"! Fanaticism aside, magic numbers are a code smell that indicates some refactoring might be called for. So instead of "3" you might want write the code as `if(length > minDaysForMailStoppage)` but that instantly raises the question of where to define that variable. You might define it at the top of the **VacationPrep()** method, but that hardly accomplishes anything. A poor programmer might make it a readonly variable inside House, but most would place it inside **Mail**, thus:

```
        public void VacationPrep(int length) {
          if(length > Mail.minDaysForMailStoppage {
            System.Console.WriteLine("Stopping the mail");
            myMail.Active = false;
           }

…etc…
```

and

```
class Mail{
   static readonly int MIN_DAYS_FOR_STOP = 3;
   public int minDaysForStoppage{
     get { return MIN_DAYS_FOR_STOP; }
   }
…etc…
```

House.VacationPrep() is still acceptably cohesive, although it's increased in complexity.

So now your user comes to you and says "Well, we only want to turn on the sprinkler if the vacation is during the growing season. And if it's during the winter, we need to

bring the plants inside, in case there's a frost." If programming were a linear process, you'd follow the same pattern you did for the mail stoppage rule and you might end up with:

```
//<Example>
//<Class>Vacation3.cs</Class>
using System;

namespace Vacation1{
    class House {
        Mail myMail = new Mail();
        Garden myGarden = new Garden();

        public void VacationPrep(DateTime startDate, int
length) {
            System.Console.WriteLine("Preparing the house for
vacation");
            if(length > Mail.minDaysBeforeStoppage){
                System.Console.WriteLine("Stopping the mail");
                myMail.Active = false;
            }
            int month = startDate.Month;
            Season season = Garden.SeasonFor(month);
            switch(season){
            case Season.GrowingSeason :
                System.Console.WriteLine("Setting the
automatic sprinker");
                myGarden.SprinklerSet = true;
                break;
            case Season.Winter :
                System.Console.WriteLine("Bringing the plants
inside");
                foreach(Plant p in myGarden.Plants){
                    if(p.Outside){
                        p.BringInside();
                    }
                }
                break;
            default:
                System.Console.WriteLine("Leaving the garden
alone");
                break;
            }
            System.Console.WriteLine("House is prepared");
        }

        public static void Main() {
            House myHouse = new House();
            myHouse.VacationPrep(DateTime.Now, 4);
```

```
    }
}

class Mail {
    static readonly int MIN_DAYS_BEFORE_STOPPAGE = 3;
    public static int minDaysBeforeStoppage{
        get { return MIN_DAYS_BEFORE_STOPPAGE; }
    }
    bool active;
    public bool Active{
        get { return active;}
        set { active = value;}
    }
}

class Garden {
    bool sprinklerOn;
    public bool SprinklerSet{
        get { return sprinklerOn;}
        set { sprinklerOn = value;}
    }

    Plant[] plants = new Plant[]{
        new Geranium(),
        new Ficus()
    };

    public Plant[] Plants{
        get { return plants; }
    }

    public static Season SeasonFor(int month){
        if(month >= 3 && month <= 10)
            return Season.GrowingSeason;
        if(month == 10)
            return Season.Dormant;
        return Season.Winter;
    }
}

enum Season{ GrowingSeason, Dormant, Winter }

class Plant{
    protected bool isOutside = true;

    public bool Outside{
        get { return isOutside; }
        set { isOutside = value; }
    }
```

```
        public void BringInside(){
            System.Console.WriteLine("Bringing a plant
inside");
            this.Outside = false;
        }
    }
    class Geranium : Plant{ }
    class Ficus : Plant{
        public Ficus(){
            this.Outside = false;
        }
    }
}
```

Now, **House.VacationPrep()** is passed both a length and a starting date. It determines the month of the vacation, uses **Garden.SeasonFor(int month)** to determine in what Season the vacation will happen, and takes the necessary steps depending on that.

As a procedural method, **VacationPrep()** would still be considered fairly cohesive. Everything it does has something to do with preparing the house for vacation, it's just turning out to be a more complex task than it was on our first iteration. But from an object standpoint, something's gone awry.

The most egregious problem is clearly the call inside the conditional inside the loop inside the switch. Oh, I need to explain where that is? That's part of the problem – deeply nested control structures are inherently complex. I'm referring to these lines:

```
if(p.Outside){
    p.BringInside();
}
```

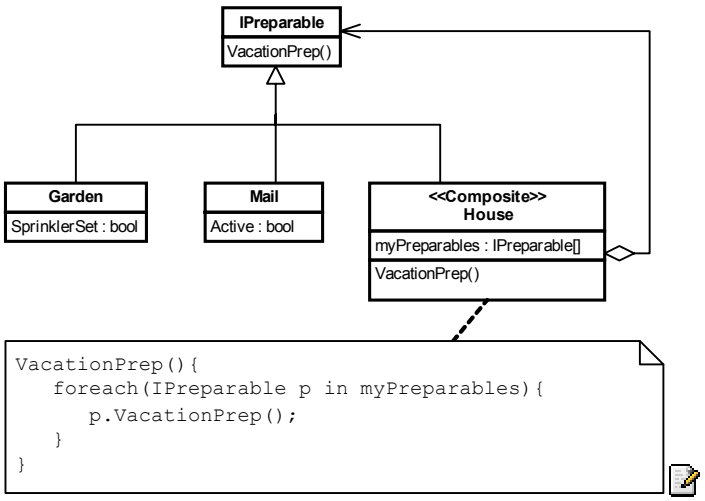
The execution of this block is based on the state of a referenced object (the **Plant p**), but the block only contains calls to that same object. In other words, both the "guard condition" and the consequence are working on external objects. Although one could argue that **House.VacationPrep()** is still fairly cohesive, this block indicates that **Plant.BringInside()** is not cohesive, because it does not contain the logically necessary check to make sure that only plants that are currently outside are brought inside. Thus, we'll change **Plant.BringInside()** to:

```
public void BringInsideIfNecessary(){
   if(this.Outside){
      System.Console.WriteLine("Bringing a plant inside");
      this.Outside = false;
   }
}
```

Something's wrong with this. A single method which does more than one thing is one type of cohesion mistake, two methods that do essentially the same thing is another. The public **Outside** property of **Plant** has a **set** method. Our newly written **BringInsideIfNecessary()** does nothing that isn't logically done inside that **set**

method, with the exception of printing something to the Console. If the distinction between printing to the console or not is important, then maybe we could use a parameter

The same people who mistakenly believe that Simulation architectures are the only "true" object-oriented architectures will say that this code is already wrong. They will, very predictably, draw this diagram:



and write this code:

```
//<Example>
//<Class>Vacation2.cs</Class>
namespace Vacation2{
    interface IPreparable{
        void VacationPrep();
    }

    class House : IPreparable{
        IPreparable[] myPreparables = {
            new Mail(),
            new Garden()
        };

        public void VacationPrep() {
            System.Console.WriteLine("Preparing the house for
vacation");
            foreach(IPreparable p in myPreparables){
                p.VacationPrep();
            }
            System.Console.WriteLine("House is prepared");
        }

        public static void Main() {
            House myHouse = new House();
            myHouse.VacationPrep();
```

```
        }
    }

    class Mail : IPreparable{
        bool active;
        public bool Active{
            get { return active;}
            set { active = value;}
        }

        public void VacationPrep(){
            System.Console.WriteLine("Mail deactivating
itself");
            this.active = false;
        }
    }

    class Garden : IPreparable {
        bool sprinklerOn;
        public bool SprinklerSet{
            get { return sprinklerOn;}
            set { sprinklerOn = value;}
        }

        public void VacationPrep(){
            System.Console.WriteLine("Garden turning on the
sprinkler");
            sprinklerOn = true;
        }
    }
}
//</Example>
```

There's nothing wrong with the code in **House.VacationPrep()** – it's cohesive and it's not complex. There's no reason, at this point, to

Sometime in your career, you're going to work on a project where every time you make a change, it causes something else to break. It will be a miserable, soul-deadening and interminable period. It will eventually end, but only because the company cancels the project or goes bankrupt.

These terrible experiences occur because it is impossible to isolate a change. There is only one possible cause for this: the area you are attempting to change is *coupled* with another area of the program. Okay, there's a small-but-finite possibility that the problem is caused by an underlying operating system or hardware flaw that's triggered by some non-programmatic aspect of the work you're doing (such flaws are

properly called “bugs” and most programmers encounter them a couple of times per decade. Programmatic mistakes are called “defects” and happen, oh, about twenty times per day).

Coupling, or dependence, is the degree to which a method or class relies on other methods or classes to accomplish its task. Coupling lowers method complexity (by relying on something else to do a calculation), but increases design complexity.

Coupling has a complex relationship with cohesion. A cohesive method does one and only one process, a cohesive class encapsulates logically related behavior and data. For all but the rarest classes, the “and only one” and “logically related” constraints require the use of other classes. So splitting a single task into two may increase both coherence and coupling and be of unarguable benefit. However, splitting and re-splitting (and re-re-splitting) a task will often backfire due to context demands.

he serious problem with much software is the prevalence of large methods that rely on deeply nested loops and conditionals for flow control. Good methods should be so straightforward that they require no internal comments. For instance:

```
Cost totalPrice(Product[] products){
   Cost sum = new Cost();
   foreach(Product p in products){
     sum += p.Cost;
   }
   return sum;
}
```

This method still needs method documentation – that it doesn’t apply taxes, that it may return a Cost whose value is 0 (assuming that’s how the Cost() constructor initializes) if the products parameter is empty, perhaps even that it requires products to be non-null (although it’s better to assume that all methods throw a NullReferenceException when passed any null parameters and document when that *isn’t* the case). But it doesn’t need internal comments (“Initialize a Cost object called ‘sum’ that will hold the results”? Please!).

Internal comments are only needed when your code is complex. Don’t write complex code. Write boring code that solves interesting problems. Cyclomatic complexity, the measure of code complexity, was introduced in chapter #3?#. One of the characteristics of good object-oriented programs is that their methods have less cyclomatic complexity than procedural programs. How can this be so, if the two programs solve the same problem? Some sample code might explain the difference.

Design Is As Design Does

6a: Hiding the Implementation

A primary consideration in object-oriented design is "separating the things that change from the things that stay the same."

This is particularly important for libraries. The user (*client programmer*) of that library must be able to rely on the part they use, and know that they won't need to rewrite code if a new version of the library comes out. On the flip side, the library creator must have the freedom to make modifications and improvements with the certainty that the client programmer's code won't be affected by those changes.

This can be achieved through convention. For example, the library programmer must agree to not remove existing methods when modifying a class in the library, since that would break the client programmer's code. The reverse situation is thornier, however. In the case of a data member, how can the library creator know which data members have been accessed by client programmers? This is also true with methods that are only part of the implementation of a class, and not meant to be used directly by the client programmer. But what if the library creator wants to rip out an old implementation and put in a new one? Changing any of those members might break a client programmer's code. Thus the library creator is in a strait jacket and can't change anything.

To solve this problem, C# provides *access specifiers* to allow the library creator to say what is available to the client programmer and what is not. The levels of access control from "most access" to "least access" are **public**, **protected**, "friendly" (which has no keyword), and **private**. From the previous paragraph you might think that, as a library designer, you'll want to keep everything as "private" as possible, and expose only the methods that you want the client programmer to use. This is exactly right, even though it's often counterintuitive for people who program in other languages (especially C) and are used to accessing everything without restriction. By the end of this chapter you should be convinced of the value of access control in C#.

The concept of a library of components and the control over who can access the components of that library is not complete, however. There's still the question of how the components are bundled together into a cohesive library unit. This is controlled with the **namespace** keyword in C#, and the access specifiers are affected by whether a class is in the same package or in a separate package. So to begin this

chapter, you'll learn how library components are placed into packages. Then you'll be able to understand the complete meaning of the access specifiers.

The namespace unit

A package is what you get when you use the **using** keyword to bring in an entire library, such as

```
using System.Collections;
```

This brings in the entire utility library that's part of the standard .NET Framework SDK distribution. Since, for example, the class **ArrayList** is in **java.util**, you can now either specify the full name **java.util.ArrayList** (which you can do without the **import** statement), or you can simply say **ArrayList** (because of the **import**).

If you want to bring in a single class, you can name that class in the **import** statement

```
import java.util.ArrayList;
```

Now you can use **ArrayList** with no qualification. However, none of the other classes in **java.util** are available.

The reason for all this importing is to provide a mechanism to manage "name spaces." The names of all your class members are insulated from each other. A method **f( )** inside a class **A** will not clash with an **f( )** that has the same signature (argument list) in class **B**. But what about the class names? Suppose you create a **stack** class that is installed on a machine that already has a **stack** class that's written by someone else? With Java on the Internet, this can happen without the user knowing it, since classes can get downloaded automatically in the process of running a Java program.

This potential clashing of names is why it's important to have complete control over the name spaces in C#, and to be able to create a completely unique name regardless of the constraints of the Internet.

So far, most of the examples in this book have existed in a single file and have been designed for local use, and haven't bothered with package names. (In this case the class name is placed in the "default package.") This is certainly an option, and for simplicity's sake this approach will be used whenever possible throughout the rest of this book. However, if you're planning to create libraries or programs that are friendly to other C# programs on the same machine, you must think about preventing class name clashes.

When you create a source-code file for C#, it's commonly called a *compilation unit* (sometimes a *translation unit*). Each compilation unit must have a name ending in **.cs**, and inside the compilation unit there can be one or more **public** classes. Tk

Creating unique package names

tk....

Now this package name can be used as an umbrella name space for the following two files:

```
//compiled with csc /target:library
namespace thinkingincsharp{
    public class ArrayList{
        public ArrayList(){

System.Console.WriteLine("thinkingincsharp.ArrayList");
        }
    }
```

...tk...

```
using thinkingincsharp;

namespace usesanother{
    class UsesSpecialized{
        public static void Main(){
            ArrayList al = new ArrayList();
            //Can still explicitly reference other
            System.Collections.ArrayList realList = new
System.Collections.ArrayList();
            realList.Add("Oh! It's a real collection class!");
        }
    }
```

...tk...

Collisions

What happens if two libraries are imported via * and they include the same names? For example, suppose a program does this:

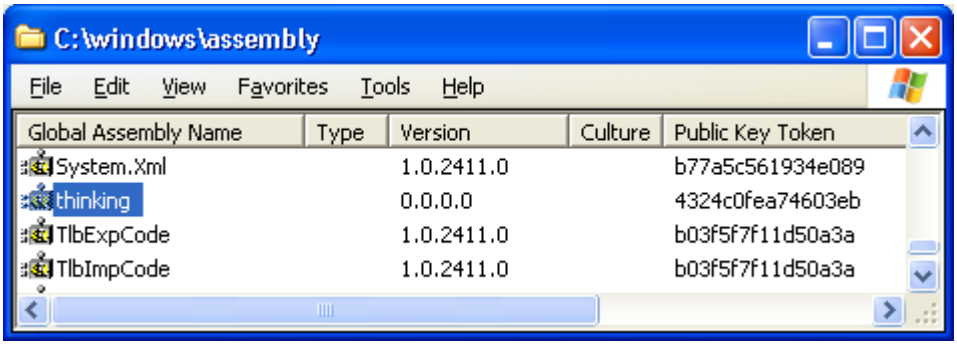
```
using Net.ThinkingIn.Util;
using System.Collections;
```

tk

```
d:\tic\chap5\thinking>csc /t:module W.cs
Microsoft (R) Visual C# Compiler Version 7.00.9254 [CLR
version v1.0.2914]
Copyright (C) Microsoft Corp 2000-2001. All rights reserved.

D:\tic\chap5\thinking>dir
 Volume in drive D is MOVIES
```

```
 Volume Serial Number is 2B37-1907

 Directory of D:\tic\chap5\thinking

10/23/2001  08:37 AM    <DIR>          .
10/23/2001  08:37 AM    <DIR>          ..
10/23/2001  08:35 AM               285 W.cs
10/23/2001  08:38 AM             2,048 W.netmodule
               2 File(s)          2,333 bytes
               2 Dir(s)  10,613,563,392 bytes free


D:\tic\chap5\thinking>sn -k key.snk

Microsoft (R) .NET Framework Strong Name Utility  Version
1.0.2914.16
Copyright (C) Microsoft Corp. 1998-2001. All rights reserved.

Key pair written to key.snk

D:\tic\chap5\thinking>dir
 Volume in drive D is MOVIES
 Volume Serial Number is 2B37-1907

 Directory of D:\tic\chap5\thinking

10/23/2001  08:37 AM    <DIR>          .
10/23/2001  08:37 AM    <DIR>          ..
10/23/2001  08:35 AM               285 W.cs
10/23/2001  08:38 AM             2,048 W.netmodule
10/23/2001  08:39 AM               596 key.snk
               4 File(s)          6,001 bytes
               2 Dir(s)  10,613,547,008 bytes free

D:\tic\chap5\thinking>al /out:thinking.dll /keyfile:key.snk
W.netmodule
Microsoft (R) Assembly Linker Version 7.00.9254 [CLR version
v1.0.2914]
Copyright (C) Microsoft Corp 2000-2001. All rights reserved.
```

```
D:\tic\chap5\thinking>dir
 Volume in drive D is MOVIES
 Volume Serial Number is 2B37-1907

 Directory of D:\tic\chap5\thinking

10/23/2001  08:37 AM    <DIR>          .
10/23/2001  08:37 AM    <DIR>          ..
10/23/2001  08:35 AM               285 W.cs
10/23/2001  08:38 AM             2,048 W.netmodule
10/23/2001  08:39 AM             3,072 W.dll
10/23/2001  08:39 AM               596 key.snk
10/23/2001  08:42 AM             3,584 thinking.dll
               5 File(s)          9,585 bytes
               2 Dir(s)  10,613,276,672 bytes free

D:\tic\chap5\thinking>gacutil /i thinking.dll

Microsoft (R) .NET Global Assembly Cache Utility.  Version
1.0.2914.16
Copyright (C) Microsoft Corp. 1998-2001. All rights reserved.

Assembly successfully added to the cache

D:\tic\chap5\thinking>start c:\windows\assembly
```



@todo: Figure out how to reference it in others (I know I can do it from the command line compile, but I can't figure out how to do it transparently)

You can use this shorthand to print a **string** either with a newline (**P.rintln( )**) or without a newline (**P.rint( )**).

Using imports to change behavior

...tk... The program will no longer print assertions. Here's an example:

```
//<example>
///<chapter>05</chapter>
///<program>TestAssert.cs</program>
// Demonstrating the assertion tool.
// Comment or uncomment the following to change assertion
behavior:
//#define ASSERT
using System.Diagnostics;

public class Assert{
    private static void PErr(string s){
        System.Console.WriteLine(s);
    }
    [Conditional("ASSERT")] public static void True(bool exp){
         if(!exp) PErr("Assertion failed");
    }
    [Conditional("ASSERT")] public static void False(bool
exp){
         if(exp) PErr("Assertion failed");
    }
    [Conditional("ASSERT")] public static void True(bool exp,
string msg){
        if(!exp) PErr(msg);
    }
    [Conditional("ASSERT")] public static void False(bool exp,
string msg){
        if(exp) PErr(msg);
    }
}

public class TestAssert {
public static void Main() {
    Assert.True((2 + 2) == 5);
    Assert.False((1 + 1) == 2);
    Assert.True((2 + 2) == 5, "2 + 2 == 5");
    Assert.False((1 + 1) == 2, "1 +1 != 2");
  }
} //</example>
```

By changing the package that's imported, you change your code from the debug version to the production version. This technique can be used for any kind of conditional code.

C#'s access specifiers

When used, the Java access specifiers **public**, **protected,** and **private** are placed in front of each definition for each member in your class, whether it's a field or a method. Each access specifier controls the access for only that particular definition. This is a distinct contrast to C++, in which the access specifier controls all the definitions following it until another access specifier comes along.

One way or another, everything has some kind of access specified for it. In the following sections, you'll learn all about the various types of access, starting with the default access.

"Friendly"

What if you give no access specifier at all, as in all the examples before this chapter? The default access has no keyword, but it is commonly referred to as "friendly." It means that all the other classes in the current package have access to the friendly member, but to all the classes outside of this package the member appears to be **private**. Since a compilation unit—a file—can belong only to a single package, all the classes within a single compilation unit are automatically friendly with each other. Thus, friendly elements are also said to have *package access*.

Friendly access allows you to group related classes together in a package so that they can easily interact with each other. When you put classes together in a package (thus granting mutual access to their friendly members; i.e., making them "friends") you "own" the code in that package. It makes sense that only code you own should have friendly access to other code you own. You could say that friendly access gives a meaning or a reason for grouping classes together in a package. In many languages the way you organize your definitions in files can be willy-nilly, but in Java you're compelled to organize them in a sensible fashion. In addition, you'll probably want to exclude classes that shouldn't have access to the classes being defined in the current package.

The class controls which code has access to its members. There's no magic way to "break in." Code from another package can't show up and say, "Hi, I'm a friend of **Bob**'s!" and expect to see the **protected**, friendly, and **private** members of **Bob**. The only way to grant access to a member is to:

1. Make the member **public**. Then everybody, everywhere, can access it.
2. Make the member friendly by leaving off any access specifier, and put the other classes in the same package. Then the other classes can access the member.
3. As you'll see in Chapter 6, when inheritance is introduced, an inherited class can access a **protected** member as well as a **public** member (but not **private**

members). It can access friendly members only if the two classes are in the same package. But don't worry about that now.

4. Provide "accessor/mutator" methods (also known as "get/set" methods) that read and change the value. This is the most civilized approach in terms of OOP, and it is fundamental to JavaBeans, as you'll see in Chapter 13.

public: interface access

When you use the **public** keyword, it means that the member declaration that immediately follows **public** is available to everyone, in particular to the client programmer who uses the library. Suppose you define a package **dessert** containing the following compilation unit:

```
//: c05:dessert:Cookie.java
// Creates a library.
package c05.dessert;

public class Cookie {
  public Cookie() {
   System.out.println("Cookie constructor");
  }
  void bite() { System.out.println("bite"); }
} ///:~
```

Remember, **Cookie.java** must reside in a subdirectory called **dessert**, in a directory under **c05** (indicating Chapter 5 of this book) that must be under one of the CLASSPATH directories. Don't make the mistake of thinking that Java will always look at the current directory as one of the starting points for searching. If you don't have a '**.**' as one of the paths in your CLASSPATH, Java won't look there.

Now if you create a program that uses **Cookie**:

```
//: c05:Dinner.java
// Uses the library.
import c05.dessert.*;

public class Dinner {
  public Dinner() {
   System.out.println("Dinner constructor");
  }
  public static void main(String[] args) {
    Cookie x = new Cookie();
    //! x.bite(); // Can't access
  }
} ///:~
```

you can create a **Cookie** object, since its constructor is **public** and the class is **public**. (We'll look more at the concept of a **public** class later.) However, the **bite( )**

member is inaccessible inside **Dinner.java** since **bite( )** is friendly only within package **dessert**.

The default package

You might be surprised to discover that the following code compiles, even though it would appear that it breaks the rules:

```
//: c05:Cake.java
// Accesses a class in a
// separate compilation unit.

class Cake {
  public static void main(String[] args) {
    Pie x = new Pie();
    x.f();
  }
} ///:~
```

In a second file, in the same directory:

```
//: c05:Pie.java
// The other class.

class Pie {
  void f() { System.out.println("Pie.f()"); }
} ///:~
```

You might initially view these as completely foreign files, and yet **Cake** is able to create a **Pie** object and call its **f( )** method! (Note that you must have '.' in your CLASSPATH in order for the files to compile.) You'd typically think that **Pie** and **f( )** are friendly and therefore not available to **Cake**. They *are* friendly—that part is correct. The reason that they are available in **Cake.java** is because they are in the same directory and have no explicit package name. Java treats files like this as implicitly part of the "default package" for that directory, and therefore friendly to all the other files in that directory.

private: you can't touch that!

The **private** keyword means that no one can access that member except that particular class, inside methods of that class. Other classes in the same package cannot access **private** members, so it's as if you're even insulating the class against yourself. On the other hand, it's not unlikely that a package might be created by several people collaborating together, so **private** allows you to freely change that member without concern that it will affect another class in the same package.

The default "friendly" package access often provides an adequate amount of hiding; remember, a "friendly" member is inaccessible to the user of the package. This is nice, since the default access is the one that you normally use (and the one that you'll get if you forget to add any access control). Thus, you'll typically think about access

for the members that you explicitly want to make **public** for the client programmer, and as a result, you might not initially think you'll use the **private** keyword often since it's tolerable to get away without it. (This is a distinct contrast with C++.) However, it turns out that the consistent use of **private** is very important, especially where multithreading is concerned. (As you'll see in Chapter 14.)

Here's an example of the use of **private**:

```
//: c05:IceCream.java
// Demonstrates "private" keyword.

class Sundae {
  private Sundae() {}
  static Sundae makeASundae() {
    return new Sundae();
  }
}

public class IceCream {
  public static void main(String[] args) {
    //! Sundae x = new Sundae();
    Sundae x = Sundae.makeASundae();
  }
} ///:~
```

This shows an example in which **private** comes in handy: you might want to control how an object is created and prevent someone from directly accessing a particular constructor (or all of them). In the example above, you cannot create a **Sundae** object via its constructor; instead you must call the **makeASundae( )** method to do it for you[4].

Any method that you're certain is only a "helper" method for that class can be made **private,** to ensure that you don't accidentally use it elsewhere in the package and thus prohibit yourself from changing or removing the method. Making a method **private** guarantees that you retain this option.

The same is true for a **private** field inside a class. Unless you must expose the underlying implementation (which is a much rarer situation than you might think), you should make all fields **private**. However, just because a reference to an object is **private** inside a class doesn't mean that some other object can't have a **public** reference to the same object. (See Appendix A for issues about aliasing.)

[4] There's another effect in this case: Since the default constructor is the only one defined, and it's **private**, it will prevent inheritance of this class. (A subject that will be introduced in Chapter 6.)

protected: "sort of friendly"

The **protected** access specifier requires a jump ahead to understand. First, you should be aware that you don't need to understand this section to continue through this book up through inheritance (Chapter 6). But for completeness, here is a brief description and example using **protected**.

The **protected** keyword deals with a concept called *inheritance*, which takes an existing class and adds new members to that class without touching the existing class, which we refer to as the *base class*. You can also change the behavior of existing members of the class. To inherit from an existing class, you say that your new class **extends** an existing class, like this:

```
class Foo extends Bar {
```

The rest of the class definition looks the same.

If you create a new package and you inherit from a class in another package, the only members you have access to are the **public** members of the original package. (Of course, if you perform the inheritance in the *same* package, you have the normal package access to all the "friendly" members.) Sometimes the creator of the base class would like to take a particular member and grant access to derived classes but not the world in general. That's what **protected** does. If you refer back to the file **Cookie.java**, the following class *cannot* access the "friendly" member:

```
//: c05:ChocolateChip.java
// Can't access friendly member
// in another class.
import c05.dessert.*;

public class ChocolateChip extends Cookie {
  public ChocolateChip() {
   System.out.println(
     "ChocolateChip constructor");
  }
  public static void main(String[] args) {
    ChocolateChip x = new ChocolateChip();
    //! x.bite(); // Can't access bite
  }
} ///:~
```

One of the interesting things about inheritance is that if a method **bite( )** exists in class **Cookie**, then it also exists in any class inherited from **Cookie**. But since **bite( )** is "friendly" in a foreign package, it's unavailable to us in this one. Of course, you could make it **public**, but then everyone would have access and maybe that's not what you want. If we change the class **Cookie** as follows:

```
public class Cookie {
  public Cookie() {
    System.out.println("Cookie constructor");
```

```
  }
  protected void bite() {
    System.out.println("bite");
  }
}
```

then **bite( )** still has "friendly" access within package **dessert**, but it is also accessible to anyone inheriting from **Cookie**. However, it is *not* **public**.

protected also gives package access – that is, other classes in the same package may access **protected** elements.

Interface and implementation

Access control is often referred to as *implementation hiding*. Wrapping data and methods within classes in combination with implementation hiding is often called *encapsulation*[5]. The result is a data type with characteristics and behaviors.

Access control puts boundaries within a data type for two important reasons. The first is to establish what the client programmers can and can't use. You can build your internal mechanisms into the structure without worrying that the client programmers will accidentally treat the internals as part of the interface that they should be using.

This feeds directly into the second reason, which is to separate the interface from the implementation. If the structure is used in a set of programs, but client programmers can't do anything but send messages to the **public** interface, then you can change anything that's *not* **public** (e.g., "friendly," **protected**, or **private**) without requiring modifications to client code.

We're now in the world of object-oriented programming, where a **class** is actually describing "a class of objects," as you would describe a class of fishes or a class of birds. Any object belonging to this class will share these characteristics and behaviors. The class is a description of the way all objects of this type will look and act.

In the original OOP language, Simula-67, the keyword **class** was used to describe a new data type. The same keyword has been used for most object-oriented languages. This is the focal point of the whole language: the creation of new data types that are more than just boxes containing data and methods.

The class is the fundamental OOP concept in C#. It is one of the keywords that will *not* be set in bold in this book—it becomes annoying with a word repeated as often as "class."

---

[5] However, people often refer to implementation hiding alone as encapsulation.

For clarity, you might prefer a style of creating classes that puts the **public** members at the beginning, followed by the **protected**, friendly, and **private** members. The advantage is that the user of the class can then read down from the top and see first what's important to them (the **public** members, because they can be accessed outside the file), and stop reading when they encounter the non-**public** members, which are part of the internal implementation:

```
public class X {
  public void Pub1( ) { /* . . . */ }
  public void Pub2( ) { /* . . . */ }
  public void Pub3( ) { /* . . . */ }
  private void priv1( ) { /* . . . */ }
  private void priv2( ) { /* . . . */ }
  private void priv3( ) { /* . . . */ }
  private int i;
  // . . .
}
```

This will make it only partially easier to read because the interface and implementation are still mixed together. That is, you still see the source code—the implementation—because it's right there in the class. In addition, the comment documentation somewhat lessens the importance of code readability by the client programmer. Displaying the interface to the consumer of a class is really the job of the *class browser*, a tool whose job is to look at all the available classes and show you what you can do with them (i.e., what members are available) in a useful fashion. Microsoft's Visual Studio .NET is the first, but not the only, tool to provide a class browser for C#.

Class access

In C#, the access specifiers can also be used to determine which classes *within* a library will be available to the users of that library. If you want a class to be available to a client programmer, you place the **public** keyword somewhere before the opening brace of the class body. This controls whether the client programmer can even create an object of the class.

To control the access of a class, the specifier must appear before the keyword **class**. Thus you can say:

```
public class Widget {
```

Now if the name of your library is **Mylib** any client programmer can access **Widget** by saying

```
using Mylib;
Widget w;
```

What if you've got a class inside **Mylib** that you're just using to accomplish the tasks performed by **Widget** or some other **public** class in **Mylib**? You don't want to go to

the bother of creating documentation for the client programmer, and you think that sometime later you might want to completely change things and rip out your class altogether, substituting a different one. To give you this flexibility, you need to ensure that no client programmers become dependent on your particular implementation details hidden inside **Mylib**. To accomplish this, you just leave the **public** keyword off the class, in which case it becomes friendly. (That class can be used only within that package.)

Note that a class cannot be **private** (that would make it accessible to no one but the class), or **protected**. So you have only two choices for class access: "friendly" or **public**. If you don't want anyone else to have access to that class, you can make all the constructors **private**, thereby preventing anyone but you, inside a **static** member of the class, from creating an object of that class[6]. Here's an example:

```
//<example>
///<chapter>c05</chapter>
///<program>Lunch.cs</program>
// Demonstrates class access specifiers.
// Make a class effectively private
// with private constructors:

class Soup {
  private Soup() {}
  // (1) Allow creation via static method:
  public static Soup MakeSoup() {
    return new Soup();
  }
  // (2) Create a static object and
  // return a reference upon request.
  // (The "Singleton" pattern):
  private static Soup ps1 = new Soup();
  public static Soup Access() {
    return ps1;
  }
  public void F() {}
}

class Sandwich { // Uses Lunch
  void F() { new Lunch(); }
}

// Only one public class allowed per file:
public class Lunch {
  void Test() {
    // Can't do this! Private constructor:
```

[6] You can also do it by inheriting (Chapter 6) from that class.

```
    Soup priv1 = new Soup();
    //! Soup priv2 = Soup.MakeSoup();
    Sandwich f1 = new Sandwich();
    Soup.Access().F();
  }
} //</example>
```

Up to now, most of the methods have been returning either **void** or a primitive type, so the definition:

```
  public static Soup Access() {
    return ps1;
  }
```

might look a little confusing at first. The word before the method name (**Access**) tells what the method returns. So far this has most often been **void,** which means it returns nothing. But you can also return a reference to an object, which is what happens here. This method returns a reference to an object of class **Soup**.

The **class Soup** shows how to prevent direct creation of a class by making all the constructors **private**. Remember that if you don't explicitly create at least one constructor, the default constructor (a constructor with no arguments) will be created for you. By writing the default constructor, it won't be created automatically. By making it **private**, no one can create an object of that class. But now how does anyone use this class? The above example shows two options. First, a **static** method is created that creates a new **Soup** and returns a reference to it. This could be useful if you want to do some extra operations on the **Soup** before returning it, or if you want to keep count of how many **Soup** objects to create (perhaps to restrict their population).

The second option uses what's called a *design pattern*, which is covered in *Thinking in Patterns with Java*, downloadable at *www.BruceEckel.com*. This particular pattern is called a "singleton" because it allows only a single object to ever be created. The object of class **Soup** is created as a **static private** member of **Soup**, so there's one and only one, and you can't get at it except through the **public** method **access( )**.

As previously mentioned, if you don't put an access specifier for class access it defaults to "friendly." This means that an object of that class can be created by any other class in the package, but not outside the package. (Remember, all the files within the same directory that don't have explicit **package** declarations are implicitly part of the default package for that directory.) However, if a **static** member of that class is **public**, the client programmer can still access that **static** member even though they cannot create an object of that class.

Summary

In any relationship it's important to have boundaries that are respected by all parties involved. When you create a library, you establish a relationship with the user of that library—the client programmer—who is another programmer, but one putting together an application or using your library to build a bigger library.

Without rules, client programmers can do anything they want with all the members of a class, even if you might prefer they don't directly manipulate some of the members. Everything's naked to the world.

This chapter looked at how classes are built to form libraries; first, the way a group of classes is packaged within a library, and second, the way the class controls access to its members.

It is estimated that a C programming project begins to break down somewhere between 50K and 100K lines of code because C has a single "name space" so names begin to collide, causing an extra management overhead. In Java, the **package** keyword, the package naming scheme, and the **import** keyword give you complete control over names, so the issue of name collision is easily avoided.

There are two reasons for controlling access to members. The first is to keep users' hands off tools that they shouldn't touch; tools that are necessary for the internal machinations of the data type, but not part of the interface that users need to solve their particular problems. So making methods and fields **private** is a service to users because they can easily see what's important to them and what they can ignore. It simplifies their understanding of the class.

The second and most important reason for access control is to allow the library designer to change the internal workings of the class without worrying about how it will affect the client programmer. You might build a class one way at first, and then discover that restructuring your code will provide much greater speed. If the interface and implementation are clearly separated and protected, you can accomplish this without forcing the user to rewrite their code.

Access specifiers in Java give valuable control to the creator of a class. The users of the class can clearly see exactly what they can use and what to ignore. More important, though, is the ability to ensure that no user becomes dependent on any part of the underlying implementation of a class. If you know this as the creator of the class, you can change the underlying implementation with the knowledge that no client programmer will be affected by the changes because they can't access that part of the class.

When you have the ability to change the underlying implementation, you can not only improve your design later, but you also have the freedom to make mistakes. No matter how carefully you plan and design you'll make mistakes. Knowing that it's

relatively safe to make these mistakes means you'll be more experimental, you'll learn faster, and you'll finish your project sooner.

The public interface to a class is what the user *does* see, so that is the most important part of the class to get "right" during analysis and design. Even that allows you some leeway for change. If you don't get the interface right the first time, you can *add* more methods, as long as you don't remove any that client programmers have already used in their code.

Exercises

7: Reusing classes

What is inheritance? Why does it help? Inheritance in C#: declaring derived classes, protected and protected internal components, virtual and override methods, abstract classes, sealed classes, hiding inherited components, versioning, lack of inheritance in structs. What are alternatives to inheritance for composing complex structures? How should we decide between composition and aggregation? What are the different types of design patterns?

One of the most compelling features about object orientation is code reuse. But to be revolutionary, you've got to be able to do a lot more than copy code and change it.

That's the approach used in procedural languages like C, and it hasn't worked very well. Like everything in C#, the solution revolves around the class. You reuse code by creating new classes, but instead of creating them from scratch, you use existing classes that someone has already built and debugged.

The trick is to use the classes without soiling the existing code. In this chapter you'll see two ways to accomplish this. The first is quite straightforward: You simply create objects of your existing class inside the new class. This is called *composition,* because the new class is composed of objects of existing classes. You're simply reusing the functionality of the code, not its form.

The second approach is more subtle. It creates a new class as a *type of* an existing class. You literally take the form of the existing class and add code to it without modifying the existing class. This magical act is called *inheritance*, and the compiler does most of the work. Inheritance is one of the cornerstones of object-oriented programming, and has additional implications that will be explored in Chapter #inheritance#.

It turns out that much of the syntax and behavior are similar for both composition and inheritance (which makes sense because they are both ways of making new types from existing types). In this chapter, you'll learn about these code reuse mechanisms.

Composition syntax

Until now, composition has been used quite frequently. You simply place object references inside new classes. For example, suppose you'd like an object that holds several **string** objects, a couple of primitives, and an object of another class. For the nonprimitive objects, you put references inside your new class, but you define the primitives directly:

```
//:c06:SprinklerSystem.cs
```

```
// Composition for code reuse.

class WaterSource {
  private string s;
  WaterSource() {
    System.Console.WriteLine("WaterSource()");
    s = "Constructed";
  }
  public override string ToString() { return s; }
}

public class SprinklerSystem {
  private string valve1, valve2, valve3, valve4;
  WaterSource source;
  int i;
  float f;
  void Print() {
    System.Console.WriteLine("valve1 = " + valve1);
    System.Console.WriteLine("valve2 = " + valve2);
    System.Console.WriteLine("valve3 = " + valve3);
    System.Console.WriteLine("valve4 = " + valve4);
    System.Console.WriteLine("i = " + i);
    System.Console.WriteLine("f = " + f);
    System.Console.WriteLine("source = " + source);
  }
  public static void Main() {
    SprinklerSystem x = new SprinklerSystem();
    x.Print();
  }
} ///:~
```

One of the methods defined in **WaterSource** is special: **ToString( )**. You will learn later that every object has a **ToString( )** method, and it's called in special situations when the compiler wants a **string** but it's got one of these objects. So in the expression:

```
System.Console.WriteLine("source = " + source);
```

the compiler sees you trying to add a **string** object ("**source =** ") to a **WaterSource**. This doesn't make sense to it, because you can only "add" a **string** to another s**tring**, so it says "I'll turn **source** into a **string** by

calling **ToString( )**!" After doing this it can combine the two **string**s and pass the resulting **String** to **System.Console.WriteLine( )**. Any time you want to allow this behavior with a class you create you need only write a **ToString( )** method.

At first glance, you might assume—C# being as safe and careful as it is—that the compiler would automatically construct objects for each of the references in the above code; for example, calling the default constructor for **WaterSource** to initialize **source**. The output of the print statement is in fact:

```
valve1 = null
valve2 = null
valve3 = null
valve4 = null
i = 0
f = 0.0
source = null
```

Primitives that are fields in a class are automatically initialized to zero, as noted in Chapter #initialization#. But the object references are initialized to **null**, and if you try to call methods for any of them you'll get an exception. It's actually pretty good (and useful) that you can still print them out without throwing an exception.

It makes sense that the compiler doesn't just create a default object for every reference because that would incur unnecessary overhead in many cases. If you want the references initialized, you can do it:

1. At the point the objects are defined. This means that they'll always be initialized before the constructor is called.
2. In the constructor for that class.
3. Right before you actually need to use the object. This is often called *lazy initialization*. It can reduce overhead in situations where the object doesn't need to be created every time.

All three approaches are shown here:

```
//:c06:Bath.cs
// Constructor initialization with composition.
```

```
class Soap {
  private string s;
  internal Soap() {
    System.Console.WriteLine("Soap()");
    s = "Constructed";
  }
  public override string ToString() { return s; }
}

public class Bath {
  private string
    // Initializing at point of definition:
    //@todo: Change to show arg constructor
    s1 = "Happy",
    s2 = "Happy",
    s3, s4;
  Soap castille;
  int i;
  float toy;
  Bath() {
    System.Console.WriteLine("Inside Bath()");
    s3 = "Joy";
    i = 47;
    toy = 3.14f;
    castille = new Soap();
  }
  void Print() {
    // Delayed initialization:
    if(s4 == null)
      s4 = "Joy";
    System.Console.WriteLine("s1 = " + s1);
    System.Console.WriteLine("s2 = " + s2);
    System.Console.WriteLine("s3 = " + s3);
    System.Console.WriteLine("s4 = " + s4);
    System.Console.WriteLine("i = " + i);
    System.Console.WriteLine("toy = " + toy);
    System.Console.WriteLine("castille = " +
castille);
  }
  public static void Main() {
```

```
    Bath b = new Bath();
    b.Print();
  }
} ///:~
```

Note that in the **Bath** constructor a statement is executed before any of the initializations take place. When you don't initialize at the point of definition, there's still no guarantee that you'll perform any initialization before you send a message to an object reference—except for the inevitable run-time exception.

Here's the output for the program:

```
Inside Bath()
Soap()
s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy
i = 47
toy = 3.14
castille = Constructed
```

When **Print( )** is called it fills in **s4** so that all the fields are properly initialized by the time they are used.

Inheritance syntax

Inheritance is an integral part of C# (and OOP languages in general). It turns out that you're always doing inheritance when you create a class, because unless you explicitly inherit from some other class, you implicitly inherit from C#'s standard root class **object**.

The syntax for composition is obvious, but to perform inheritance there's a distinctly different form. When you inherit, you say "This new class is like that old class." You state this in code by giving the name of the class as usual, but before the opening brace of the class body, put a colon followed by the name of the *base class*. When you do this, you automatically get all the data members and methods in the base class. Here's an example:

```
//:c06:Detergent.cs
///Compile with: "/main:Detergent"
// Inheritance syntax & properties.

internal class Cleanser {
  private string s = "Cleanser";
  public void Append(string a) { s += a; }
  public void Dilute() { Append(" dilute()"); }
  public void Apply() { Append(" apply()"); }
  virtual public void Scrub() { Append(" scrub()"); }
  public void Print() { System.Console.WriteLine(s); }
  public static void Main(string[] args) {
    Cleanser x = new Cleanser();
    x.Dilute(); x.Apply(); x.Scrub();
    x.Print();
  }
}

internal class Detergent : Cleanser {
  // Change a method:
  override public void Scrub() {
    Append(" Detergent.scrub()");
    base.Scrub(); // Call base-class version
  }
  // Add methods to the interface:
  public void Foam() { Append(" Foam()"); }
  // Test the new class:
  new public static void Main(string[] args) {
    Detergent x = new Detergent(args);
    x.Dilute();
    x.Apply();
    x.Scrub();
    x.Foam();
    x.Print();
    System.Console.WriteLine("Testing base class:");
    Cleanser.Main();
  }
} ///:~
```

This demonstrates a number of features. First, both **Cleanser** and **Detergent** contain a **Main( )** method. You can create a **Main( )** for each

one of your classes, but if you do so, the compiler will generate an error, saying that you are defining multiple entry points. You can choose which **Main()** you want to have associated with the assembly by using the /Main:Classname switch. Thus, if you compile the above with **csc Detergent.cs /Main:Cleanser**, the output will be:

```
Cleanser dilute() apply() scrub()
```

While if compiled with **csc Detergent.cs /Main:Detergent**, the result is:

```
Cleanser dilute() apply() Detergent.scrub() scrub()
Foam()
Testing base class:
Cleanser dilute() apply() scrub()
```

This technique of putting a **Main( )** in each class can sometimes help with testing, when you just want to write a quick little program to make sure your methods are working the way you intend them to. But for general testing purposes, you should use a unit-testing framework (see Chapter #testing#). You don't need to remove the **Main( )** when you're finished testing; you can leave it in for later testing.

Here, you can see that **Detergent.Main( )** calls **Cleanser.Main( )** explicitly, passing it the same arguments from the command line (however, you could pass it any **string** array).

It's important that all of the methods in **Cleanser** are **public**. Remember that if you leave off any access specifier the member defaults to **private**, which allows access only to the very class in which the field or method is defined. So to plan for inheritance, as a general rule leave fields **private**, but make all methods **public**. (**protected** members also allow access by derived classes; you'll learn about this later.) Of course, in particular cases you must make adjustments, but this is a useful guideline.

Note that **Cleanser** has a set of methods in its interface: **Append( )**, **Dilute( )**, **Apply( )**, **Scrub( )**, and **Print( )**. Because **Detergent** is *derived from* **Cleanser** it automatically gets all these methods in its interface, even though you don't see them all explicitly defined in

Detergent. You can think of inheritance, then, as *reusing the interface*. (The implementation also comes with it, but that part isn't the primary point.)

As seen in **Scrub( )**, it's possible to take a method that's been defined in the base class and modify it. In this case, you might want to call the method from the base class inside the new version. But inside **Scrub( )** you cannot simply call **Scrub( )**, since that would produce a recursive call, which isn't what you want. To solve this problem C# has the keyword **base** that refers to the "base class" (also called the "superclass") from which the current class has been inherited. Thus the expression **base.Scrub( )** calls the base-class version of the method **Scrub( )**.

When inheriting you're not restricted to using the methods of the base class. You can also add new methods to the derived class exactly the way you put any method in a class: just define it. The method **Foam( )** is an example of this.

In **Detergent.Main( )** you can see that for a **Detergent** object you can call all the methods that are available in **Cleanser** as well as in **Detergent** (i.e., **Foam( )**).

Initializing the base class

Since there are now two classes involved—the base class and the derived class—instead of just one, it can be a bit confusing to try to imagine the resulting object produced by a derived class. From the outside, it looks like the new class has the same interface as the base class and maybe some additional methods and fields. But inheritance doesn't just copy the interface of the base class. When you create an object of the derived class, it contains within it a *subobject* of the base class. This subobject is the same as if you had created an object of the base class by itself. It's just that, from the outside, the subobject of the base class is wrapped within the derived-class object.

Of course, it's essential that the base-class subobject be initialized correctly and there's only one way to guarantee that: perform the initialization in the constructor, by calling the base-class constructor, which has all the appropriate knowledge and privileges to perform the base-class initialization. C# automatically inserts calls to the base-class

constructor in the derived-class constructor. The following example shows this working with three levels of inheritance:

```
//:c06:Cartoon.cs
// Constructor calls during inheritance.
//@todo: check if access modifiers can be made more
private
public class Art {
  internal Art() {
    System.Console.WriteLine("Art constructor");
  }
}

public class Drawing : Art {
  internal Drawing() {
    System.Console.WriteLine("Drawing constructor");
  }
}

public class Cartoon : Drawing {
  internal Cartoon() {
    System.Console.WriteLine("Cartoon constructor");
  }
  public static void Main() {
    Cartoon x = new Cartoon();
  }
} ///:~
```

The output for this program shows the automatic calls:

```
Art constructor
Drawing constructor
Cartoon constructor
```

You can see that the construction happens from the base "outward," so the base class is initialized before the derived-class constructors can access it.

Even if you don't create a constructor for **Cartoon( )**, the compiler will synthesize a default constructor for you that calls the base class constructor.

Constructors with arguments

The above example has default constructors; that is, they don't have any arguments. It's easy for the compiler to call these because there's no question about what arguments to pass. If your class doesn't have default arguments, or if you want to call a base-class constructor that has an argument, you must explicitly write the calls to the base-class constructor using the **base** keyword and the appropriate argument list:

```
//:c06:Chess.cs
// Inheritance, constructors and arguments.

public class Game {
  internal Game(int i) {
    System.Console.WriteLine("Game constructor");
  }
}

public class BoardGame : Game {
  internal BoardGame(int i) : base(i) {
    System.Console.WriteLine("BoardGame constructor");
  }
}

public class Chess : BoardGame {
  internal Chess() : base(11){
    System.Console.WriteLine("Chess constructor");
  }
  public static void Main() {
    Chess x = new Chess();
  }
}///:~
```

If you don't call the base-class constructor in **BoardGame( )**, the compiler will complain that it can't find a constructor of the form **Game( )**.

Catching base constructor exceptions

As just noted, the compiler forces you to place the base-class constructor call before even the body of the derived-class constructor. As you'll see in Chapter #Exceptions#, this also prevents a derived-class constructor from

catching any exceptions that come from a base class. This can be inconvenient at times.

```
//:c06:Dome.cs
using System;

class Dome{
    public Dome(){
       throw new InvalidOperationException();
    }
}

class Brunelleschi : Dome{
    public Brunelleschi(){
        System.Console.WriteLine("Ingenious
Vaulting");
    }

    public static void Main(){
        try{
            new Brunelleschi();
        }catch(Exception ex){
            System.Console.WriteLine(ex);
        }
    }
}///:~
```

prints:

```
System.InvalidOperationException: Operation is not
valid due to the current state of the object.
   at Dome..ctor()
   at Brunelleschi.Main()
```

Combining composition and inheritance

It is very common to use composition and inheritance together. The following example shows the creation of a more complex class, using both

inheritance and composition, along with the necessary constructor initialization:

```
//:c06:PlaceSetting.cs
// Combining composition & inheritance.

class Plate {
  internal Plate(int i) {
    System.Console.WriteLine("Plate constructor");
  }
}

class DinnerPlate : Plate {
  internal DinnerPlate(int i) : base(i) {
    System.Console.WriteLine(
      "DinnerPlate constructor");
  }
}

class Utensil {
  internal Utensil(int i) {
    System.Console.WriteLine("Utensil constructor");
  }
}

class Spoon : Utensil {
  internal Spoon(int i) : base(i) {
    System.Console.WriteLine("Spoon constructor");
  }
}

class Fork : Utensil {
  internal Fork(int i) : base(i) {
    System.Console.WriteLine("Fork constructor");
  }
}

class Knife : Utensil {
  internal Knife(int i) : base(i) {
    System.Console.WriteLine("Knife constructor");
  }
```

```
}

// A cultural way of doing something:
class Custom {
  internal Custom(int i) {
    System.Console.WriteLine("Custom constructor");
  }
}

class PlaceSetting : Custom {
  Spoon sp;
  Fork frk;
  Knife kn;
  DinnerPlate pl;
  PlaceSetting(int i) : base(i + 1) {
    sp = new Spoon(i + 2);
    frk = new Fork(i + 3);
    kn = new Knife(i + 4);
    pl = new DinnerPlate(i + 5);
    System.Console.WriteLine(
      "PlaceSetting constructor");
  }
  public static void Main (){
    PlaceSetting x = new PlaceSetting(9);
  }
}///:~
```

While the compiler forces you to initialize the base classes, and requires that you do it right at the beginning of the constructor, it doesn't watch over you to make sure that you initialize the member objects, so you must remember to pay attention to that.

Guaranteeing proper cleanup

You may recall from chapter #initialization# that although C# has a destructor, we said that the proper way to guarantee that an object cleans up after itself involved the **IDisposable** interface, implementing the method **Dispose(),** and wrapping the "valuable resource" in a **using** block. At the time, we deferred a discussion of how it worked, but with an understanding of inheritance, it starts to becomes clear. (Although

understanding how the **using** block works will require an understanding of Excepions, which is coming in Chapter #exceptions# (“.)

Consider an example of a computer-aided design system that draws pictures on the screen:

```
//:c06:CADSystem.cs
// Ensuring proper cleanup.
using System;

class Shape : IDisposable {
  internal Shape(int i) {
    System.Console.WriteLine("Shape constructor");
  }
  public virtual void Dispose() {
    System.Console.WriteLine("Shape disposed");
  }
}

class Circle : Shape {
  internal Circle(int i) : base(i) {
    System.Console.WriteLine("Drawing a Circle");
  }
  public override void Dispose() {
    System.Console.WriteLine("Erasing a Circle");
    base.Dispose();
  }
}

class Triangle : Shape {
  internal Triangle(int i) : base(i) {
    System.Console.WriteLine("Drawing a Triangle");
  }
  public override void Dispose() {
    System.Console.WriteLine("Erasing a Triangle");
    base.Dispose();
  }
}

class Line : Shape {
  private int start, end;
```

```
  internal Line(int start, int end) : base(start){
    this.start = start;
    this.end = end;
    System.Console.WriteLine("Drawing a Line: " +
           start + ", " + end);
  }
  public override void Dispose() {
    System.Console.WriteLine("Erasing a Line: " +
           start + ", " + end);
    base.Dispose();
  }
}

class CADSystem : Shape {
  private Circle c;
  private Triangle t;
  private Line[] lines = new Line[10];
  CADSystem(int i) : base(i + 1){
    for(int j = 0; j < 10; j++)
      lines[j] = new Line(j, j*j);
    c = new Circle(1);
    t = new Triangle(1);
    System.Console.WriteLine("Combined constructor");
  }
  public override void Dispose() {
    System.Console.WriteLine("CADSystem.Dispose()");
    // The order of cleanup is the reverse
    // of the order of initialization
    t.Dispose();
    c.Dispose();
    for(int i = lines.Length - 1; i >= 0; i--)
      lines[i].Dispose();
    base.Dispose();
  }
  public static void Main() {
    CADSystem x = new CADSystem(47);
    using(x){
      // Code and exception handling...
    }
    System.Console.WriteLine("Using block left");
  }
```

```
}///:~
```
Everything in this system is some kind of **Shape** (which itself is a kind of **object** since it's implicitly inherited from the root class and which implements an *interface* called IDisposable). Each class redefines **Shape**'s **Dispose( )** method in addition to calling the base-class version of that method using **base**. The specific **Shape** classes—**Circle**, **Triangle** and **Line**—all have constructors that "draw," although any method called during the lifetime of the object could be responsible for doing something that needs cleanup. Each class has its own **Dispose( )** method to restore nonmemory things back to the way they were before the object existed.

In **Main( )**, you can see the **using** keyword in action. A **using** block takes an **IDisposable** as an argument. When execution leaves the block (even if an exception is thrown), **IDisposable.Dispose()** is called. But because we have implemented **Dispose()** in **Shape** and all the classes derived from it, inheritance kicks in and the appropriate **Dispose()** method is called. In this case, the using block has a **CADSystem**. It's **Dispose()** method calls, in turn, the **Dispose()** method of the objects which comprise it.

Note that in your cleanup method you must also pay attention to the calling order for the base-class and member-object cleanup methods in case one subobject depends on another. In general, you should follow the same form that is imposed by a C++ compiler on its destructors: First perform all of the cleanup work specific to your class, in the reverse order of creation. (In general, this requires that base-class elements still be viable.) Then call the base-class **Dispose** method, as demonstrated here.

There can be many cases in which the cleanup issue is not a problem; you just let the garbage collector do the work. But when you must do it explicitly, diligence and attention is required.

Order of garbage collection

There's not much you can rely on when it comes to garbage collection. The garbage collector may not be called until your program exits. If it is called, it can reclaim objects in any order it wants. It's best to not rely on garbage collection for anything but memory reclamation. If you have "valuable

resources" which need explicit cleanup, always initialize them as late as possible, and dispose of them as soon as you can.

Choosing composition vs. inheritance

Both composition and inheritance allow you to place subobjects inside your new class. You might wonder about the difference between the two, and when to choose one over the other.

Composition is generally used when you want the features of an existing class inside your new class, but not its interface. That is, you embed an object so that you can use it to implement functionality in your new class, but the user of your new class sees the interface you've defined for the new class rather than the interface from the embedded object. For this effect, you embed **private** objects of existing classes inside your new class.

Sometimes it makes sense to allow the class user to directly access the composition of your new class; that is, to make the member objects **public**. The member objects use implementation hiding themselves, so this is a safe thing to do. When the user knows you're assembling a bunch of parts, it makes the interface easier to understand. A **car** object is a good example:

```
//:c06:Car.cs
// Composition with public objects.
//@todo: Good place to talk about enforced access
visibility (Engine can't be internal if Engine is
public in Car)

public class Engine {
  public void start() {}
  public void rev() {}
  public void stop() {}
}
```

```
public class Wheel {
  public void inflate(int psi) {}
}

public class Window {
  public void rollup() {}
  public void rolldown() {}
}

public class Door {
  public Window window = new Window();
  public void open() {}
  public void close() {}
}

public class Car {
  public Engine engine = new Engine();
  public Wheel[] wheel = new Wheel[4];
  public Door left = new Door(),
       right = new Door(); // 2-door
  public Car() {
    for(int i = 0; i < 4; i++)
      wheel[i] = new Wheel();
  }
  public static void Main() {
    Car car = new Car();
    car.left.window.rollup();
    car.wheel[0].inflate(72);
  }
} ///:~
```

@todo: Good place to talk about indexers?

Because the composition of a car is part of the analysis of the problem (and not simply part of the underlying design), making the members **public** assists the client programmer's understanding of how to use the class and requires less code complexity for the creator of the class. However, keep in mind that this is a special case and that in general you should make fields **private**.

When you inherit, you take an existing class and make a special version of it. In general, this means that you're taking a general-purpose class and specializing it for a particular need. With a little thought, you'll see that it would make no sense to compose a car using a vehicle object—a car doesn't contain a vehicle, it *is* a vehicle. The *is-a* relationship is expressed with inheritance, and the *has-a* relationship is expressed with composition.

protected

Now that you've been introduced to inheritance, the keyword **protected** finally has meaning. In an ideal world, **private** members would always be hard-and-fast **private**, but in real projects there are times when you want to make something hidden from the world at large and yet allow access for members of derived classes. The **protected** keyword is a nod to pragmatism. It says "This is **private** as far as the class user is concerned, but available to anyone who inherits from this class." @todo: Introduce **internal** in discussion of namespaces

The best tack to take is to leave the data members **private**—you should always preserve your right to change the underlying implementation. You can then allow controlled access to inheritors of your class through **protected** methods:

```
//:c06:Orc.cs
// The protected keyword.

public class Villain {
  private int i;
  protected int read() { return i; }
  protected void set(int ii) { i = ii; }
  public Villain(int ii) { i = ii; }
  public int value(int m) { return m*i; }
}

public class Orc : Villain {
  private int j;
  public Orc(int jj) :base(jj) {  j = jj; }
  public void change(int x) { set(x); }
```

```
} ///:~
```

You can see that **change( )** has access to **set( )** because it's **protected**.

Incremental development

One of the advantages of inheritance is that it supports *incremental development* by allowing you to introduce new code without causing bugs in existing code. This also isolates new bugs inside the new code. By inheriting from an existing, functional class and adding data members and methods (and redefining existing methods), you leave the existing code—that someone else might still be using—untouched and unbugged. If a bug happens, you know that it's in your new code, which is much shorter and easier to read than if you had modified the body of existing code.

It's rather amazing how cleanly the classes are separated. You don't even need the source code for the methods in order to reuse the code. At most, you just import a package. (This is true for both inheritance and composition.)

It's important to realize that program development is an incremental process, just like human learning. You can do as much analysis as you want, but you still won't know all the answers when you set out on a project. You'll have much more success—and more immediate feedback—if you start out to "grow" your project as an organic, evolutionary creature, rather than constructing it all at once like a glass-box skyscraper.

Although inheritance for experimentation can be a useful technique, at some point after things stabilize you need to take a new look at your class hierarchy with an eye to collapsing it into a sensible structure. Remember that underneath it all, inheritance is meant to express a relationship that says "This new class is a *type of* that old class." Your program should not be concerned with pushing bits around, but instead with creating and manipulating objects of various types to express a model in the terms that come from the problem space.

Upcasting

The most important aspect of inheritance is not that it provides methods for the new class. It's the relationship expressed between the new class and the base class. This relationship can be summarized by saying "The new class *is a type of* the existing class."

This description is not just a fanciful way of explaining inheritance—it's supported directly by the language. As an example, consider a base class called **Instrument** that represents musical instruments, and a derived class called **Wind**. Because inheritance means that all of the methods in the base class are also available in the derived class, any message you can send to the base class can also be sent to the derived class. If the **Instrument** class has a **Play( )** method, so will **Wind** instruments. This means we can accurately say that a **Wind** object is also a type of **Instrument**. The following example shows how the compiler supports this notion:

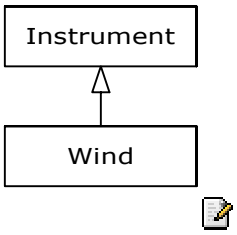
```
//:c06:Wind.cs
// Inheritance & upcasting.

public class Instrument {
  public void play() {}
  static internal void tune(Instrument i) {
    // ...
    i.play();
  }
}

// Wind objects are instruments
// because they have the same interface:
public class Wind : Instrument {
  public static void Main() {
    Wind flute = new Wind();
    Instrument.tune(flute); // Upcasting
  }
} ///:~
```

What's interesting in this example is the **Tune( )** method, which accepts an **Instrument** reference. However, in **Wind**.**Main( )** the **Tune( )**

method is called by giving it a **Wind** reference. Given that C# is particular about type checking, it seems strange that a method that accepts one type will readily accept another type, until you realize that a **Wind** object is also an **Instrument** object, and there's no method that **Tune( )** could call for an **Instrument** that isn't also in **Wind**. Inside **Tune( )**, the code works for **Instrument** and anything derived from **Instrument**, and the act of converting a **Wind** reference into an **Instrument** reference is called *upcasting*.

Why "upcasting"?

The reason for the term is historical, and based on the way class inheritance diagrams have traditionally been drawn: with the root at the top of the page, growing downward. (Of course, you can draw your diagrams any way you find helpful.) The inheritance diagram for **Wind.java** is then:



Casting from derived to base moves *up* on the inheritance diagram, so it's commonly referred to as *upcasting*. Upcasting is always safe because you're going from a more specific type to a more general type. That is, the derived class is a superset of the base class. It might contain more methods than the base class, but it must contain *at least* the methods in the base class. The only thing that can occur to the class interface during the upcast is that it can lose methods, not gain them. This is why the compiler allows upcasting without any explicit casts or other special notation.

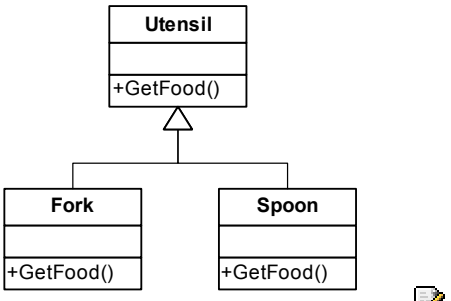
You can also perform the reverse of upcasting, called *downcasting*, but this involves a dilemma that is the subject of Chapter 12.

Composition vs. inheritance revisited

In object-oriented programming, the most likely way that you'll create and use code is by simply packaging data and methods together into a class, and using objects of that class. You'll also use existing classes to build new classes with composition. Less frequently, you'll use inheritance. So although inheritance gets a lot of emphasis while learning OOP, it doesn't mean that you should use it everywhere you possibly can. On the contrary, you should use it sparingly, only when it's clear that inheritance is useful. One of the clearest ways to determine whether you should use composition or inheritance is to ask whether you'll ever need to upcast from your new class to the base class. If you must upcast, then inheritance is necessary, but if you don't need to upcast, then you should look closely at whether you need inheritance. The next chapter (polymorphism) provides one of the most compelling reasons for upcasting, but if you remember to ask "Do I need to upcast?" you'll have a good tool for deciding between composition and inheritance.

Explicit Overloading Only

Some of C#'s most notable departures from the object-oriented norm are the barriers it places on the road to overloading functionality. In most object-oriented languages, if you have classes Fork and Spoon that descends from Utensil, a base method **GetFood**, and two implementations of it, you just declare the method in the base and have identical signatures in the descending classes:



In Java, this would look like:

```
class Utensil{
   public void GetFood(){ //…}
}

class Fork extends Utensil{
   public void GetFood(){ System.out.println("Spear");
}
}

class Spoon extends Utensil{
   public void GetFood(){ System.out.println("Scoop");
}
}
```

In C#, you have to jump through a bit of a hoop; methods for which overloading is intended must be declared **virtual** and the overloading method must be declared as an **override**. To get the desired structure would look like this:

```
class Utensil{
   public virtual void GetFood(){ //…}
}

class Fork extends Utensil{
   public override void GetFood(){
     System.out.println("Spear");
   }
}

class Spoon extends Utensil{
   public override void GetFood(){
      System.out.println("Scoop");
   }
}
```

This is a behavior that stems from Microsoft's experience with "DLL Hell" and thoughts about a world in which object-oriented components are the building blocks of very large systems. Imagine that you are using Java and using a 3rd-party "Kitchen" component that includes the base class of

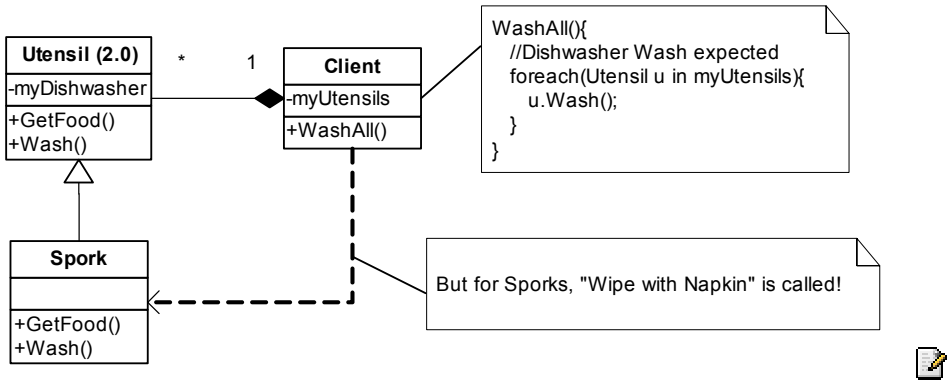
Utensil, but you customize it to use that staple of dorm life – the Spork. But in addition to implementing **GetFood()**, you add a dorm-like method **Wash()**:

```
//Spork.java
class Spork extends Utensil{
   public void GetFood(){
      System.out.println("Spear OR Scoop!");
   }

   public void Wash(){
      System.out.println("Wipe with napkin");
   }
}
```

Of course, since **Wash** isn't implemented in **Utensil**, you could only "wash" a spork (which is just as well, considering the unhygienic nature of the implementation). So the problem happens when the 3rd-party Kitchen component vendor releases a new version of their component, and this time they've implement a method with an identical signature to the one you wrote:

```
//Utensil.java @version: 2.0
class Utensil{
   public void GetFood(){ //… }
   public void Wash(){
     myDishwasher.add(this);
     //etc…
   }
}
```

The vendor has implemented a **Wash()** method with complex behavior involving a dishwasher. Given this new capability, people programming with Utensil v2 will have every right to assume that once **Wash()** has been called, *all* Utensils will have gone through the dishwasher. But in languages such as Java, the **Wash()** method in Spork will still be called!



It may seem highly unlikely that a new version of a base class would "just happen" to have the same name as an end-user's extension, but if you think about it, it's actually kind of surprising it doesn't happen more often, as the number of logical method names for a given category of base class is fairly limited.

In C#, the behavior in **Client**'s **WashAll()** method would work exactly the way clients expect, with **Utensil**'s dishwasher **Wash()** being called for all utensils in **myUtensils**, even if one happens to be a **Spork**.

Now let's say you come along and start working on Spork again after upgrading to the version of Utensil that has a **Wash()** method. When you compile Spork.cs, the compiler will say:

```
warning CS0108: The keyword new is required on
'Spork.Wash()' because it hides inherited member
'Utensil.Wash()'
```

At this point, calls to **Utensil.Wash()** are resolved with the dishwasher washing method, while if you have a handle to a Spork, the napkin-wiping wash method will be called.

```
//:c06:Utensil.cs

class Utensil{
    public virtual void GetFood(){ }
    public void Wash(){
```

```
        System.Console.WriteLine("Washing in a
dishwasher");
    }
}

class Fork : Utensil{
    public override void GetFood(){
        System.Console.WriteLine("Spear");
    }
}

class Spork : Utensil{
    public override void GetFood(){
        System.Console.WriteLine("Spear OR Scoop!");
    }

    public void Wash(){
        System.Console.WriteLine("Wipe with napkin");
    }
}

class Client{
    Utensil[] myUtensils;
    Client(){
        myUtensils = new Utensil[2];
        myUtensils[0] = new Spork();
        myUtensils[1] = new Fork();
    }
    public void WashAll(){
        foreach(Utensil u in myUtensils){
            u.Wash();
        }
    }

    public static void Main(){
        Client c = new Client();
        c.WashAll();
        Spork s = new Spork();
        s.Wash();
    }
}///:~
```

results in the output:

```
Washing in a dishwasher
Washing in a dishwasher
Wipe with napkin
```

In order to remove the warning that **Spork.Wash()** is hiding the newly minted **Utensil.Wash()**, we can add the keyword **new** to **Spork**'s declaration:

```
public new void Wash(){ //… etc ...
```

It's even possible for you to have entirely separate method inheritance hierarchies by declaring a method as **new virtual**. Imagine that for version 3 of the Kitchen component, they've created a new type of **Utensil**, **Silverware**, which requires polishing after cleaning. Meanwhile, you've created a new kind of **Spork**, a **SuperSpork**, which also has overridden the base **Spork.Wash()** method.

The code looks like this:

```
//:c06:Utensil2.cs

class Utensil{
    public virtual void GetFood(){ }
    public virtual void Wash(){
        System.Console.WriteLine("Washing in a
dishwasher");
    }
}

class Silverware : Utensil{
    public override void Wash(){
        base.Wash();
        System.Console.WriteLine("Polish with silver
cleaner");
    }
}

class Fork : Silverware{
    public override void GetFood(){
        System.Console.WriteLine("Spear");
    }
```

```
}

class Spork : Silverware{
    public override void GetFood(){
        System.Console.WriteLine("Spear OR Scoop!");
    }

    public new virtual void Wash(){
        System.Console.WriteLine("Wipe with napkin");
    }
}

class SuperSpork : Spork{
    public override void GetFood(){
        System.Console.WriteLine("Spear AND Scoop");

    }

    public override void Wash(){
        base.Wash();
        System.Console.WriteLine("Polish with shirt");
    }
}

class Client{
    Utensil[] myUtensils;
    Client(){
        myUtensils = new Utensil[3];
        myUtensils[0] = new Spork();
        myUtensils[1] = new Fork();
        myUtensils[2] = new SuperSpork();
    }
    public void WashAll(){
        foreach(Utensil u in myUtensils){
            u.Wash();
        }
        System.Console.WriteLine("All Utensils
washed");
    }

    public static void Main(){
```

```
        Client c = new Client();
        c.WashAll();
        Spork s = new SuperSpork();
        s.Wash();
    }
}///:~
```

Now, all of our Utensils have been replaced by Silverware and, when Client.WashAll() is called, **Silverware.Wash()** overloads **Utensil.Wash()**. (Note that **Silverware.Wash()** calls **Utensil.Wash()** using **base.Wash(),** in the same manner as base constructors can be called.) All Utensils in Client's .myUtensils array are now washed in a dishwasher and then polished. Note the declaration in Spork:

```
public new virtual void Wash(){ //etc }
```

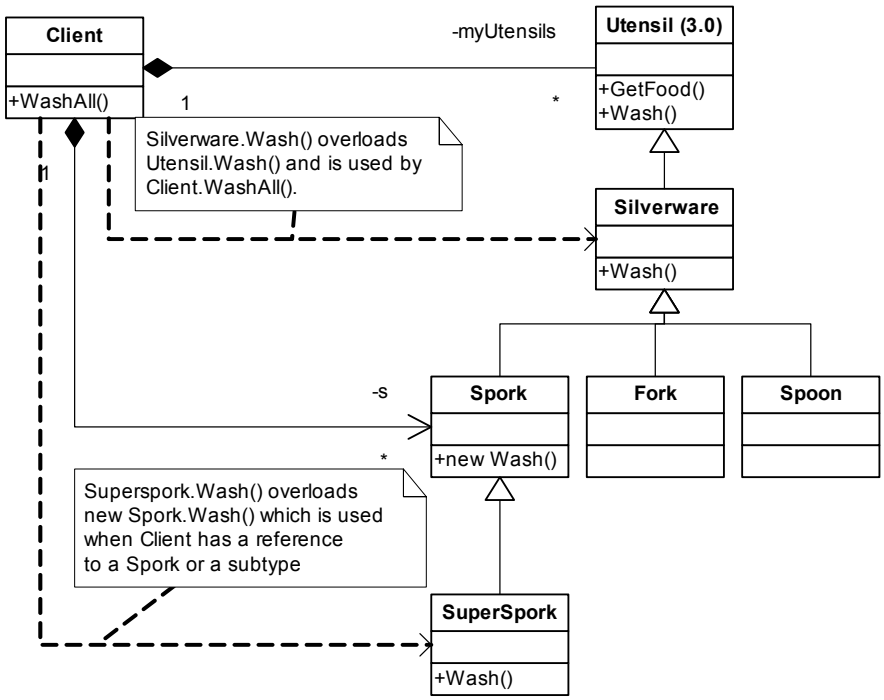
and the declaration in the newly minted **SuperSpork** class:

```
public override void Wash(){ //etc. }
```

When the Client class has a reference to a Utensil such as it does in **WashAll**() (whether the concrete type of that Utensil be a Fork, a Spoon, or a Spork), the **Wash()** method resolves to the appropriate overloaded method in **Silverware**. When, however, the client has a reference to a Spork or any Spork-subtype, the **Wash()** method resolves to whatever has overloaded **Spork**'s **new virtual Wash()**. The output looks like this:

```
Washing in a dishwasher
Polish with silver cleaner
Washing in a dishwasher
Polish with silver cleaner
Washing in a dishwasher
Polish with silver cleaner
All Utensils washed
Wipe with napkin
Polish with shirt
```

And this UML diagram shows the behavior graphically:



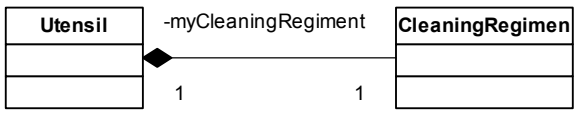
Let's say that you wanted to create a new class **SelfCleansingSuperSpork**, that overloaded both the **Wash()** method as defined in **Utensil** and the **Wash()** method as defined in **Spork**. What could you do? You cannot create a single method name that overrides both base methods. As is generally the case, when faced with a hard programming problem, the answer lies in design, not language syntax. Remember our maxim: boring code, interesting results.

One of the first things that jumps out when considering this problem is that the inheritance hierarchy is getting deep. What we're proposing is that a **SelfCleaningSuperSpork** is-a **SuperSpork** is-a **Spork** is-a **Silverware** is-a **Utensil** is-an **object**. That's six levels of hierarchy – one more than Linnaeus used to classify all living beings in 1735! It's not impossible for a design to have this many layers of inheritance, but in

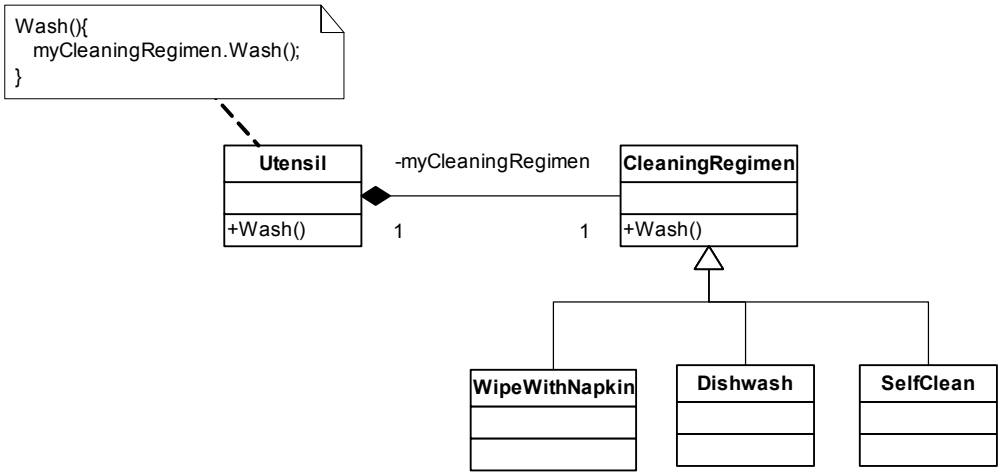
general, one should be dubious of hierarchies of more than two or three levels below **object.**

Bearing in mind that our hierarchy is getting deep, we might also notice that our names are becoming long and unnatural – **SelfCleaningSuperSpork.** While coming up with descriptive names without getting cute is one of the harder tasks in programming -- I can't tell you how many **execute(), run(),** and **query()** methods I've seen, while I've heard a story of a variable labeled **beethoven** because it's initial value happened to be 1216, the composer's birthday – something's wrong when a class name becomes a hodge-podge of adjectives. In this case, our names are being used to distinguish between two different properties – the shape of the Utensil (**Fork, Spoon, Spork,** and **SuperSpork**) and the cleaning behavior (**Silverware, Spork,** and **SelfCleaningSuperSpork**).

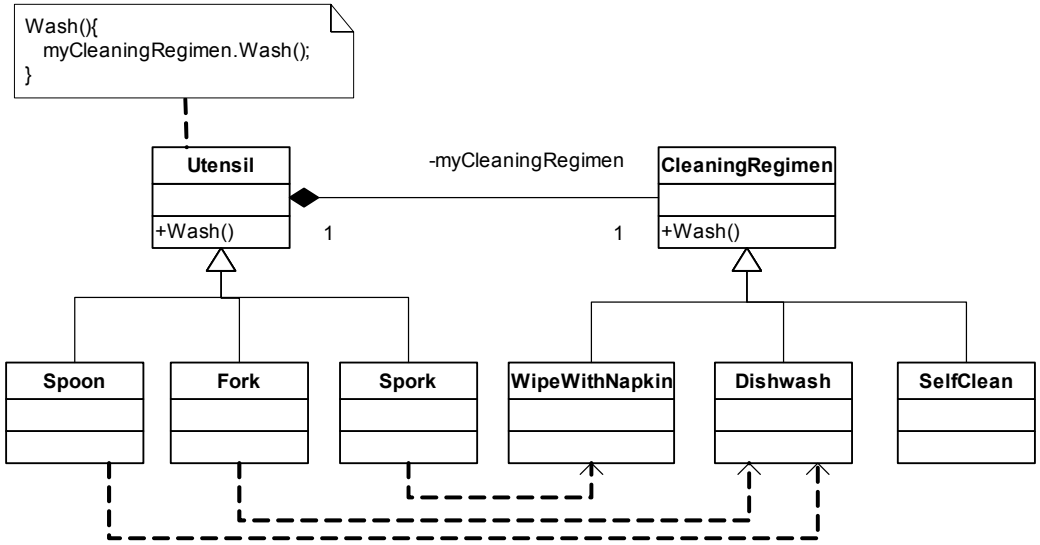
This is a clue that our design would be better using composition rather than inheritance. As is very often the case, we discover that one of the "vectors of change" is more naturally structural (the shape of the utensil) and that another is more behavioral (the cleaning regimen). We can try out the phrase "A utensil has a cleaning regimen," to see if it sounds right, which indeed it does:



When a Utensil is constructed, it has a handle to a particular type of cleaning regimen, but its Wash method doesn't have to know the specific subtype of CleaningRegimen it is using:



This is called the *Strategy* Pattern and it is, I think, the most important of all the design patterns.



This is what the code would look like:

```
//:c06:Utensil3.cs
using System;
```

```
public class Utensil{
   private CleaningRegimen myCleaningRegimen;
   protected Utensil(CleaningRegimen reg){
         myCleaningRegimen = reg;
   }

   public void Wash(){
         myCleaningRegimen.Wash();
   }

   public virtual void GetFood(){
   }
}

class Fork{
   public Fork() : base(new Dishwash()){}

   public override void GetFood(){
         System.Console.WriteLine("Spear food");
   }
}

class Spoon{
   public Spoon() : base(new Dishwash()){}
   public override void GetFood(){
         System.Console.WriteLine("Scoop food");
   }
}

class Spork{
   public Spork() : base(new WipeWithNapkin()){}
   public override void GetFood(){
         System.Console.WriteLine("Spear or scoop!");
   }
}

class CleaningRegimen{
   protected override void Wash();
}
```

```
class Dishwash : CleaningRegimen{
   protected override void Wash(){
         System.Console.WriteLine("Wash in
dishwasher");
   }
}

class WipeWithNapkin : CleaningRegimen{
   protected override void Wash(){
         System.Console.WriteLine("Wipe with napkin");
   }
}///:~
```

At this point, every type of Utensil has a particular type of CleaningRegimen associated with it, an association which is hard-coded in the constructors of the Utensil subtypes (i.e., **public Spork() : base(new WipeWithNapkin())**). However, you can see how it would be a trivial matter to totally decouple the **Utensil**'s type of **CleaningRegimen** from the constructor – you could pass in the **CleaningRegimen** from someplace else, choose it randomly, and so forth.

With this design, one can easily achieve our goal of a super utensil that combines multiple cleaning strategies:

```
class SuperSpork : Spork{
    CleaningRegimen secondRegimen;
    public SuperSpork: super(new Dishwash()){
      secondRegimen = new NapkinWash();
    }
    public override void Wash(){
      base.Wash();
      secondRegimen.Wash();
    }
}
```

In this situation, the SuperSpork now has two CleaningRegimens, the normal **myCleaningRegimen** and a new **secondRegimen**. This is the type of flexibility that you can hope to achieve by favoring aggregation over inheritance.

Our original challenge, though, involved a 3rd party Kitchen component that provided the basic design. Without access to the source code, there is no way to implement our improved design. This is one of the things that makes it hard to write components for reuse – "fully baked" components that are easy to use out of the box are often hard to customize and extend, while "construction kit" components that need to be assembled typically can sometimes take a long time to learn.

The **const** and **readonly** keywords

I know a CTO who, when reviewing code samples of potential programmers, scans for numeric constants in the code – one strike and the resume goes in the trash. I'm happy I never showed him any code of mine that did calendar math, because I don't think NUMBER_OF_DAYS_IN_WEEK is clearer than 7. Nevertheless, application code often has lots of data that never changes and C# provides two choices as to how to embody them in code.

The **const** keyword can be applied to fields of what are called the "value types": sbyte, byte, short, ushort, int, uint, long, ulong, float, double, decimal, bool, char, string, and enums. Value types differ from the more common "reference types" in that the runtime does not go through a level of indirection to reach value types.

@todo image here

const fields are evaluated at compile-time, allowing for marginal performance improvements. For instance:

```
//Number of milliseconds in a day
const long MS_PER_DAY = 1000 * 60 * 60 * 24;
```

Will be replaced at compile time with the single value 86,400,000 rather than triggering three multiplications every time it is used.

The **readonly** keyword is more general. It can be applied to any type and is evaluated once-and-only-once at runtime. Typically, **readonly** fields are initialized at either the time of class loading (in the case of static fields), or at the time of instance initialization for instance variables. It's not necessary to limit **readonly** fields to values that are essentially constant, you could use a **readonly** field for any data that, once assigned, should be invariant – a person's name or social security number, a network address or port of a host, etc.

readonly does not make an object immutable. When applied to a non-value-type object, **readonly** locks only your reference to the object, not the state of the object itself. Such an object can go through whatever state transitions are programmed into it – properties can be set, it can change its internal state based on calculations, etc. The only thing you can't do is change the reference to the object. This can be seen in this example, which demonstrates **readonly**.

```
//:c06:Composition.cs
using System;
using System.Threading;

namespace Composition {
   public class ReadOnly{
         static readonly DateTime timeOfClassLoad =
DateTime.Now;
         readonly DateTime timeOfInstanceCreation =
DateTime.Now;
         public ReadOnly() {
               System.Console.WriteLine("Class loaded
at {0}, Instance created at {1}", timeOfClassLoad,
timeOfInstanceCreation);
         }

          //used in second part of program
         static readonly ReadOnly ro = new ReadOnly();
         public int id;
         public int Id{
               get{ return id; }
               set{ id = value; }
         }
```

```
        public static void Main(){
              for(int i = 0; i < 10; i++){
                    new ReadOnly();
                    Thread.Sleep(1000);
              }
              //Can change member
              ro.Id = 5;
              System.Console.WriteLine(ro.Id);
              //! Compiler says "a static readonly
field cannot be
              // assigned to"
                //ro = new ReadOnly();
        }
}
}///:~
```

In order to demonstrate how objects created at different times will have different fields, the program uses the Thread.Sleep() method from the Threading namespace, which will be discussed at length in #multithreading#. The class **ReadOnly** contains two **readonly** fields – the static **TimeOfClassLoad** field and the instance variable **timeOfInstanceCreation**. These fields are of type **DateTime**, which is the basic .NET object for counting time. Both fields are initialized from the static DateTime property Now, which represents the system clock.

When the **Main** creates the first **ReadOnly** object and the static fields are initialized as discussed previously, **TimeOfClassLoad** is set once-and-for-all. Then, the instance variable field **timeOfInstanceCreation** is initialized. Finally, the constructor is called, and it prints the value of these two fields to the console. **Thread.Sleep(1000)** is then used to pause the program for a second (1,000 milliseconds) before creating another **ReadOnly**. The behavior of the program until this point would be no different if these fields were not declared as **readonly**, since we have made no attempt to modify the fields.

That changes in the lines below the loop. In addition to the **readonly DateTime** fields, we have a **static readonly ReadOnly** field labeled **ro**

(the class **ReadOnly** contains a reference to an instance of **ReadOnly** – a common idiom that is known as a "singleton"). We also have a property called **Id**, but note that it is not **readonly**.

(As a review of the discussion in #initialization#, you should be able to figure out how the values of **ro**'s **timeOfClassLoad** and **timeOfInstanceCreation** will relate to the first **ReadOnly** created in the **Main** loop.)

Although the reference to **ro** is read only, the line ro.Id = 5; demonstrates how it is possible to change the state of a **readonly** reference. What we can't do, though, is shown in the commented-out lines in the example – if we attempt to assign to **ro**, we'll get a compile time error.

The advantage of **readonly** over **const** is that **const**'s compile-time math is immutable. If a class PhysicalConstants had a **public const** that set the speed of light to 300,000 kilometers per second and another class used that for compile-time math (`const long KILOMETERS_IN_A_LIGHT_YEAR = PhysicalConstants.C * 3600 * 24 * DAYS_PER_YEAR`), the value of `KILOMETERS_IN_A_LIGHT_YEAR` will be based on the 300,000 value, even if the base class is updated to a more accurate value such as 299,792. This will be true until the class that defined **KILOMETERS_IN_A_LIGHT_YEAR** is recompiled with access to the updated PhysicalConstants class. If the fields were **readonly** though, the value for **KILOMETERS_IN_A_LIGHT_YEAR** would be calculated at runtime, and would not need to be recompiled to properly reflect the latest value of C. Again, this is one of those features which may not seem like a big deal to many application developers, but whose necessity is clear to Microsoft after a decade of "DLL Hell."

Sealed classes

The **readonly** and **const** keywords are used for locking down values and references that should not be changed. Because one has to declare a method as **virtual** in order to be overridden, it is easy to create methods that will not be modified at runtime. Naturally, there is a way to specify that an entire class be unmodifiable. When a class is declared as **sealed**, no one can derive from it.

There are two main reasons to make a class **sealed**. A **sealed** class is more secure from intentional or unintentional tampering. Additionally, virtual methods executed on a **sealed** class can be replaced with direct function calls, providing a slight performance increase.

```
//:c06:Jurassic.cs
// Sealing a class

class SmallBrain {}

sealed class Dinosaur {
  internal int i = 7;
  internal int j = 1;
  SmallBrain x = new SmallBrain();
  internal void F() {}
}

//! class Further : Dinosaur {}
// error: Cannot extend sealed class 'Dinosaur'

public class Jurassic {
  public static void Main() {
    Dinosaur n = new Dinosaur();
    n.F();
    n.i = 40;
    n.j++;
  }
}///:~
```

Defining the class as **sealed** simply prevents inheritance—nothing more. However, because it prevents inheritance all methods in a **sealed** class are implicitly non-**virtual**, since there's no way to override them.

@todo: C# doesn't have final arguments?

Emphasize virtual functions

It can seem sensible to make as few methods as possible **virtual** and even to declare a class as **sealed**. You might feel that efficiency is very important when using your class and that no one could possibly want to override your methods anyway. Sometimes this is true.

But be careful with your assumptions. In general, it's difficult to anticipate how a class can be reused, especially a general-purpose class. Unless you declare a method as **virtual**, you prevent the possibility of reusing your class through inheritance in some other programmer's project simply because you couldn't imagine it being used that way.

Initialization and class loading

In more traditional languages, programs are loaded all at once as part of the startup process. This is followed by initialization, and then the program begins. The process of initialization in these languages must be carefully controlled so that the order of initialization of **static**s doesn't cause trouble. C++, for example, has problems if one **static** expects another **static** to be valid before the second one has been initialized.

C# doesn't have this problem because it takes a different approach to loading. Because everything in C# is an object, many activities become easier, and this is one of them. As you will learn more fully in the next chapter, the compiled code for a set of related classes exists in their own separate file, called an assembly. That file isn't loaded until the code is needed. In general, you can say that "Class code is loaded at the point of first use." This is often not until the first object of that class is constructed, but loading also occurs when a **static** field or **static** method is accessed.

The point of first use is also where the **static** initialization takes place. All the **static** objects and the **static** code block will be initialized in textual order (that is, the order that you write them down in the class definition) at the point of loading. The **static**s, of course, are initialized only once.

Initialization with inheritance

It's helpful to look at the whole initialization process, including inheritance, to get a full picture of what happens. Consider the following code:

```
//:c06:Beetle.cs
```

```
// The full process of initialization.

class Insect {
  int i = 9;
  internal int j;
  internal Insect() {
    Prt("i = " + i + ", j = " + j);
    j = 39;
  }
  static int x1 =
    Prt("static Insect.x1 initialized");
  internal static int Prt(string s) {
    System.Console.WriteLine(s);
    return 47;
    }
    }

class Beetle : Insect {
  int k = Prt("Beetle.k initialized");
  Beetle() {
    Prt("k = " + k);
    Prt("j = " + j);
  }
  static int x2 =
    Prt("static Beetle.x2 initialized");
  public static void Main() {
    Prt("Beetle constructor");
    Beetle b = new Beetle();
  }
} ///:~
```

The output for this program is:

```
static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 47
j = 39
```

The first thing that happens when you run **Beetle** is that you try to access **Beetle.Main( )** (a **static** method), so the loader goes out and finds the compiled code for the **Beetle** class (this happens to be in an assembly called **Beetle.exe**). In the process of loading it, the loader notices that it has a base class (that's what the colon after **class Beetle** says), which it then loads. This will happen whether or not you're going to make an object of that base class. (Try commenting out the object creation to prove it to yourself.)

If the base class has a base class, that second base class would then be loaded, and so on. Next, the **static** initialization in the root base class (in this case, **Insect**) is performed, and then the next derived class, and so on. This is important because the derived-class static initialization might depend on the base class member being initialized properly.

At this point, the necessary classes have all been loaded so the object can be created. First, all the primitives in this object are set to their default values and the object references are set to **null**—this happens in one fell swoop by setting the memory in the object to binary zero. Then the base-class constructor will be called. In this case the call is automatic, but you can also specify the base-class constructor call (as the first operation in the **Beetle( )** constructor) using **base**. The base class construction goes through the same process in the same order as the derived-class constructor. After the base-class constructor completes, the instance variables are initialized in textual order. Finally, the rest of the body of the constructor is executed.

Summary

Both inheritance and composition allow you to create a new type from existing types. Typically, however, you use composition to reuse existing types as part of the underlying implementation of the new type, and inheritance when you want to reuse the interface. Since the derived class has the base-class interface, it can be *upcast* to the base, which is critical for polymorphism, as you'll see in the next chapter.

Despite the strong emphasis on inheritance in object-oriented programming, when you start a design you should generally prefer composition during the first cut and use inheritance only when it is clearly

necessary. Composition tends to be more flexible. In addition, by using the added artifice of inheritance with your member type, you can change the exact type, and thus the behavior, of those member objects at run-time. Therefore, you can change the behavior of the composed object at run-time.

Although code reuse through composition and inheritance is helpful for rapid project development, you'll generally want to redesign your class hierarchy before allowing other programmers to become dependent on it. Your goal is a hierarchy in which each class has a specific use and is neither too big (encompassing so much functionality that it's unwieldy to reuse) nor annoyingly small (you can't use it by itself or without adding functionality).

Exercises

8: Interfaces and Implementation

Inheritance is all well and good, but polymorphism is the true cornerstone of object-oriented programming. What distinguishes an OOP program is that control is performed primarily by polymorphism, *not* flow-control statements such as if. This chapter ties the previous four chapters together into a single approach to software architecture that scales from one hundred lines of code to a hundred thousand lines and beyond. Small, simple examples are used to demonstrate the techniques that are used over-and-over again to create even the most complex software systems. Structural and behavioral patterns are briefly discussed and readers are put on the path towards composing systems from groups of patterns.

Polymorphism is the next essential feature of an object-oriented programming language after data abstraction. It allows programs to be developed in the form of interacting agreements or "contracts" that specify the behavior, but not the implementation, of classes.

Polymorphism provides a dimension of separation of interface from implementation, to decouple *what* from *how*. Polymorphism allows improved code organization and readability as well as the creation of *extensible* programs that can be "grown" not only during the original creation of the project but also when new features are desired.

Encapsulation creates new data types by combining characteristics and behaviors. Implementation hiding separates the interface from the implementation by making the details **private**. This sort of mechanical organization makes ready sense to someone with a procedural programming background. But polymorphism deals with decoupling in terms of *types*. In the last chapter, you saw how inheritance allows the treatment of an object as its own type *or* its base type. This ability is critical because it allows many types (derived from the same base type) to be treated as if they were one type, and a single piece of code to work on all those different types equally. The polymorphic method call allows one type to express its distinction from another, similar type, as long as they're both derived from the same base type. This distinction is expressed through differences in behavior of the methods that you can call through the base class.

In this chapter, you'll learn about polymorphism (also called *dynamic binding* or *late binding* or *run-time binding*) starting from the basics, with simple examples that strip away everything but the polymorphic behavior of the program.

Upcasting revisited

In Chapter #Reusing your objects# you saw how an object can be used as its own type or as an object of its base type. Taking an object reference and treating it as a reference to its base type is called *upcasting*, because of the way inheritance trees are drawn with the base class at the top.

You also saw a problem arise, which is embodied in the following:

```
//:c07:Music.cs
// Inheritance & upcasting.

public class Note {
  private int value;
  private Note(int val) { value = val; }
  public static Note
    MIDDLE_C = new Note(0),
    C_SHARP  = new Note(1),
    B_FLAT   = new Note(2);
} // Etc.

public class Instrument {
  public virtual void Play(Note n) {
    System.Console.WriteLine("Instrument.Play()");
  }
}

// Wind objects are instruments
// because they have the same interface:
public class Wind : Instrument {
  // Redefine interface method:
  public override void Play(Note n) {
    System.Console.WriteLine("Wind.Play()");
  }
}

public class Music {
  public static void Tune(Instrument i) {
    // ...
    i.Play(Note.MIDDLE_C);
  }
  public static void Main() {
    Wind flute = new Wind();
    Tune(flute); // Upcasting
  }
} ///:~
```

The method **Music.Tune( )** accepts an **Instrument** reference, but also anything derived from **Instrument**. In **Main( )**, you can see this

happening as a **Wind** reference is passed to **Tune( )**, with no cast necessary. This is acceptable; the interface in **Instrument** must exist in **Wind**, because **Wind** is inherited from **Instrument**. Upcasting from **Wind** to **Instrument** may "narrow" that interface, but it cannot make it anything less than the full interface to **Instrument**.

Forgetting the object type

This program might seem strange to you. Why should anyone intentionally *forget* the type of an object? This is what happens when you upcast, and it seems like it could be much more straightforward if **Tune( )** simply takes a **Wind** reference as its argument. This brings up an essential point: If you did that, you'd need to write a new **Tune( )** for every type of **Instrument** in your system. Suppose we follow this reasoning and add **Stringed** and **Brass** instruments:

```
//:c07:Music2.cs
// Overloading instead of upcasting.

class Note {
  private int value;
  private Note(int val) { value = val; }
  public static readonly Note
    MIDDLE_C = new Note(0),
    C_SHARP = new Note(1),
    B_FLAT = new Note(2);
} // Etc.

class Instrument {
  internal virtual void Play(Note n) {
    System.Console.WriteLine("Instrument.Play()");
  }
}

class Wind : Instrument {
  internal override void Play(Note n) {
    System.Console.WriteLine("Wind.Play()");
  }
}

class Stringed : Instrument {
```

```
  internal override void Play(Note n) {
    System.Console.WriteLine("Stringed.Play()");
  }
}

class Brass : Instrument {
  internal override void Play(Note n) {
    System.Console.WriteLine("Brass.Play()");
  }
}

public class Music2 {
  internal static void Tune(Wind i) {
    i.Play(Note.MIDDLE_C);
  }
  internal static void Tune(Stringed i) {
    i.Play(Note.MIDDLE_C);
  }
  internal static void Tune(Brass i) {
    i.Play(Note.MIDDLE_C);
  }
  public static void Main() {
    Wind flute = new Wind();
    Stringed violin = new Stringed();
    Brass frenchHorn = new Brass();
    Tune(flute); // No upcasting
    Tune(violin);
    Tune(frenchHorn);
  }
} ///:~
```

This works, but there's a major drawback: You must write type-specific methods for each new **Instrument** class you add. This means more programming in the first place, but it also means that if you want to add a new method like **Tune( )** or a new type of **Instrument**, you've got a lot of work to do. Add the fact that the compiler won't give you any error messages if you forget to overload one of your methods and the whole process of working with types becomes unmanageable.

Wouldn't it be much nicer if you could just write a single method that takes the base class as its argument, and not any of the specific derived

classes? That is, wouldn't it be nice if you could forget that there are derived classes, and write your code to talk only to the base class?

That's exactly what polymorphism allows you to do. However, most programmers who come from a procedural programming background have a bit of trouble with the way polymorphism works.

The twist

The difficulty with **Music.cs** can be seen by running the program. The output is **Wind.Play( )**. This is clearly the desired output, but it doesn't seem to make sense that it would work that way. Look at the **Tune( )** method:

```
public static void tune(Instrument i) {
  // ...
  i.Play(Note.MIDDLE_C);
}
```

It receives an **Instrument** reference. So how can the compiler possibly know that this **Instrument** reference points to a **Wind** in this case and not a **Brass** or **Stringed**? The compiler can't. To get a deeper understanding of the issue, it's helpful to examine the subject of *binding*.

Method-call binding

Connecting a method call to a method body is called *binding*. When binding is performed before the program is run (by the compiler and linker, if there is one), it's called *early binding*. You might not have heard the term before because it has never been an option with procedural languages. C compilers have only one kind of method call, and that's early binding.

The confusing part of the above program revolves around early binding because the compiler cannot know the correct method to call when it has only an **Instrument** reference.

The solution is called *late binding*, which means that the binding occurs at run-time based on the type of object. Late binding is also called

dynamic binding or *run-time binding*. When a language implements late binding, there must be some mechanism to determine the type of the object at run-time and to call the appropriate method. That is, the compiler still doesn't know the object type, but the method-call mechanism finds out and calls the correct method body. The late-binding mechanism varies from language to language, but you can imagine that some sort of type information must be installed in the objects.

Obviously, since there's additional behavior at runtime, late binding is a little more time-consuming than early binding. More importantly, if a method is early bound and some other conditions are met, an optimizing compiler may decide not to make a call at all, but instead to place a copy of the method's source code directly into the source code where the call occurs. Such *inlining* may cause the resulting binary code to be a little larger, but can result in significant performance increases in tight loops, especially when the called method is small. Additionally, the contents of an early-bound method can be analyzed and additional optimizations that can never be safely applied to late-bound methods (such as aggressive *code motion* optimizations) may be possible. To give you an idea, a 2001 study[1] showed Fortran-90 running several times as fast as, and sometimes more than an order of magnitude faster than, Java on a series of math-oriented benchmarks (the authors' prototype performance-oriented Java compiler and libraries gave dramatic speedups). I ported some of the benchmarks to C# and was disappointed to see results that were very comparable to Java performance[2].

All methods declared as **virtual** or **override** in C# use late binding, otherwise, they use early binding (confirm). This is an irritation but not a big burden. There are two scenarios: either you know that you're going to override a method later on, in which case it's no big deal to add the keyword, or you discover down the road that you need to override a method that you hadn't planned on overriding, which is a significant enough design change to justify a re-examination and recompilation of the base class' code! The one thing you can't do is change the binding

---

[1] The Ninja Project, Moreira et al., CACM 44(10), Oct 2001

[2] For details, see www.thinkingin.net

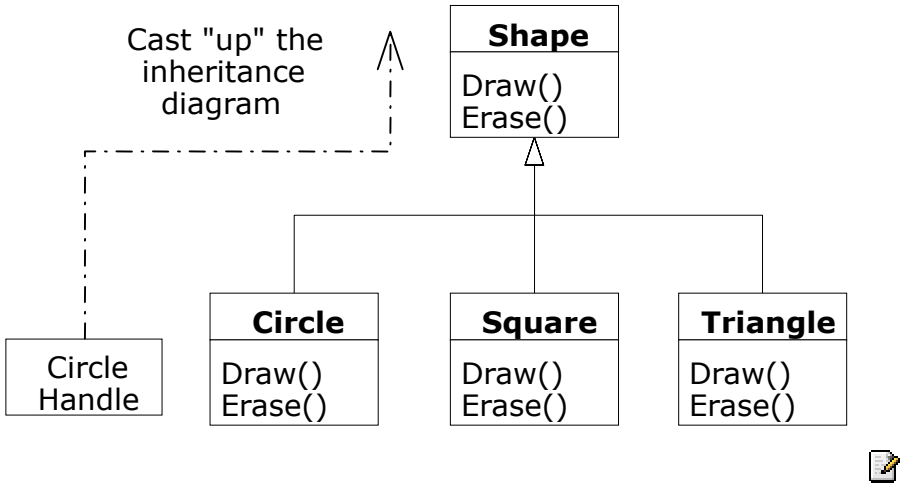
from early-bound to late-bound in a component for which you can't perform a recompile because you don't have the source code.

Producing the right behavior

Once you know that virtual method binding in C# happens polymorphically via late binding, you can write your code to talk to the base class and know that all the derived-class cases will work correctly using the same code. Or to put it another way, you "send a message to an object and let the object figure out the right thing to do."

The classic example in OOP is the "shape" example. This is commonly used because it is easy to visualize, but unfortunately it can confuse novice programmers into thinking that OOP is just for graphics programming, which is of course not the case.

The shape example has a base class called **Shape** and various derived types: **Circle**, **Square**, **Triangle**, etc. The reason the example works so well is that it's easy to say "a circle is a type of shape" and be understood. The inheritance diagram shows the relationships:



The upcast could occur in a statement as simple as:

```
Shape s = new Circle();
```

Here, a **Circle** object is created and the resulting reference is immediately assigned to a **Shape**, which would seem to be an error (assigning one type to another); and yet it's fine because a **Circle** *is* a **Shape** by inheritance. So the compiler agrees with the statement and doesn't issue an error message.

Suppose you call one of the base-class methods (that have been overridden in the derived classes):

```
s.Draw();
```

Again, you might expect that **Shape**'s **Draw( )** is called because this is, after all, a **Shape** reference—so how could the compiler know to do anything else? And yet the proper **Circle.Draw( )** is called because of late binding (polymorphism).

The following example puts it a slightly different way:

```
//:c07:Shapes.cs
// Polymorphism in C#
using System;

public class Shape {
  internal virtual void Draw() {}
  internal virtual void Erase() {}
}

class Circle : Shape {
  internal override void Draw() {
    System.Console.WriteLine("Circle.Draw()");
  }
  internal override void Erase() {
    System.Console.WriteLine("Circle.Erase()");
  }
}

class Square : Shape {
  internal override void Draw() {
    System.Console.WriteLine("Square.Draw()");
  }
  internal override void Erase() {
    System.Console.WriteLine("Square.Erase()");
```

```
  }
}

class Triangle : Shape {
  internal override void Draw() {
    System.Console.WriteLine("Triangle.Draw()");
  }
  internal override void Erase() {
    System.Console.WriteLine("Triangle.Erase()");
  }
}

public class Shapes {
   static Random rand = new Random();

  public static Shape RandShape() {
    switch(rand.Next(3)) {
      case 0: return new Circle();
      case 1: return new Square();
    case 2: return new Triangle();
    default: return null;
    }
  }
  public static void Main() {
    Shape[] s = new Shape[9];
    // Fill up the array with shapes:
    for(int i = 0; i < s.Length;i++)
      s[i] = RandShape();
    // Make polymorphic method calls:
    foreach(Shape aShape in s)
      aShape.Draw();
  }
} ///:~
```

The base class **Shape** establishes the common interface to anything inherited from **Shape**—that is, all shapes can be drawn and erased. The derived classes override these definitions to provide unique behavior for each specific type of shape.

The main class **Shapes** contains a **static** method **RandShape( )** that produces a reference to a randomly-selected **Shape** object each time you call it. Note that the upcasting happens in the **return** statements, each of

which takes a reference to a **Circle**, **Square**, or **Triangle** and sends it out of the method as the return type, **Shape**. So whenever you call this method you never get a chance to see what specific type it is, since you always get back a plain **Shape** reference.

Main( ) contains an array of **Shape** references filled through calls to **RandShape( )**. At this point you know you have **Shape**s, but you don't know anything more specific than that (and neither does the compiler). However, when you step through this array and call **Draw( )** for each one, the correct type-specific behavior magically occurs, as you can see from one output example:

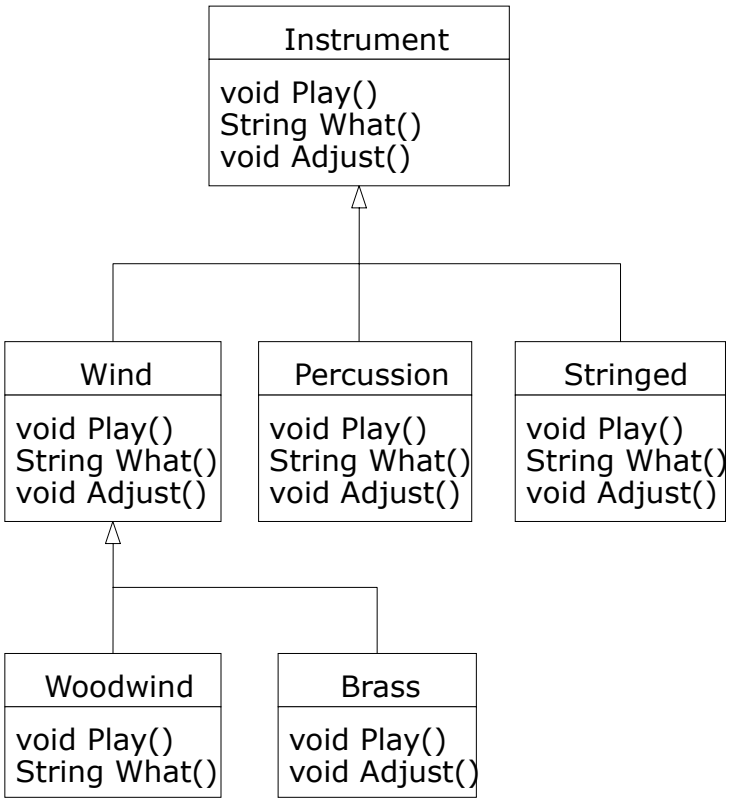
```
Circle.Draw()
Triangle.Draw()
Circle.Draw()
Circle.Draw()
Circle.Draw()
Square.Draw()
Triangle.Draw()
Square.Draw()
Square.Draw()
```

Of course, since the shapes are all chosen randomly each time, your runs will have different results. The point of choosing the shapes randomly is to drive home the understanding that the compiler can have no special knowledge that allows it to make the correct calls at compile-time. All the calls to **Draw( )** are made through dynamic binding.

Extensibility

Now let's return to the musical instrument example. Because of polymorphism, you can add as many new types as you want to the system without changing the **Tune( )** method. In a well-designed OOP program, most or all of your methods will follow the model of **Tune( )** and communicate only with the base-class interface. Such a program is *extensible* because you can add new functionality by inheriting new data types from the common base class. The methods that manipulate the base-class interface will not need to be changed at all to accommodate the new classes.

Consider what happens if you take the instrument example and add more methods in the base class and a number of new classes. Here's the diagram:



All these new classes work correctly with the old, unchanged **Tune( )** method. Even if **Tune( )** is in a separate file and new methods are added to the interface of **Instrument**, **Tune( )** works correctly without recompilation. Here is the implementation of the above diagram:

```
//:c07:Music3.cs
// An extensible program.

class Instrument {
  public virtual void Play() {
    System.Console.WriteLine("Instrument.Play()");
  }
```

```
  public virtual string What() {
    return "Instrument";
  }
  public virtual void Adjust() {}
}

class Wind : Instrument {
  public override void Play() {
    System.Console.WriteLine("Wind.Play()");
  }
  public override string What() { return "Wind"; }
  public override void Adjust() {}
}

class Percussion : Instrument {
  public override void Play() {
    System.Console.WriteLine("Percussion.Play()");
  }
  public override string What() { return "Percussion";
}
  public override void Adjust() {}
}

class Stringed : Instrument {
  public override void Play() {
    System.Console.WriteLine("stringed.Play()");
  }
  public override string What() { return "stringed"; }
  public override void Adjust() {}
}

class Brass : Wind {
  public override void Play() {
    System.Console.WriteLine("Brass.Play()");
  }
  public override void Adjust() {
    System.Console.WriteLine("Brass.Adjust()");
  }
}

class Woodwind : Wind {
```

```
  public override void Play() {
    System.Console.WriteLine("Woodwind.Play()");
  }
  public override string What() { return "Woodwind"; }
}

public class Music3 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void Tune(Instrument i) {
    // ...
    i.Play();
  }
  static void TuneAll(Instrument[] e) {
    foreach(Instrument i in e)
      Tune(i);
  }
  public static void Main() {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    TuneAll(orchestra);
  }
} ///:~
```

The new methods are **What( )**, which returns a **String** reference with a description of the class, and **Adjust( )**, which provides some way to adjust each instrument.

In **Main( )**, when you place something inside the **Instrument** array you automatically upcast to **Instrument**.

You can see that the **Tune( )** method is blissfully ignorant of all the code changes that have happened around it, and yet it works correctly. This is exactly what polymorphism is supposed to provide. Your code changes don't cause damage to parts of the program that should not be affected. Put another way, polymorphism is one of the most important techniques

that allow the programmer to "separate the things that change from the things that stay the same."

Overriding vs. overloading

Let's take a different look at the first example in this chapter. In the following program, the interface of the method **Play( )** is changed in the process of overriding it, which means that you haven't *overridden* the method, but instead *overloaded* it. The compiler allows you to overload methods so it gives no complaint. But the behavior is probably not what you want. Here's the example:

```
//:c07:WindError.cs
// Accidentally changing the interface.

public class NoteX {
  public const int
    MIDDLE_C = 0, C_SHARP = 1, C_FLAT = 2;
}

public class InstrumentX {
  public void Play(int NoteX) {
    System.Console.WriteLine("InstrumentX.Play()");
  }
}

public class WindX : InstrumentX {
  // OOPS! Changes the method interface:
  public void Play(NoteX n) {
    System.Console.WriteLine("WindX.Play(NoteX n)");
  }
}

public class WindError {
  public static void Tune(InstrumentX i) {
    // ...
    i.Play(NoteX.MIDDLE_C);
  }
  public static void Main() {
    WindX flute = new WindX();
```

```
    Tune(flute); // Not the desired behavior!
  }
} ///:~
```

There's another confusing aspect thrown in here. In **InstrumentX**, the **Play( )** method takes an **int** that has the identifier **NoteX**. That is, even though **NoteX** is a class name, it can also be used as an identifier without complaint. But in **WindX**, **Play( )** takes a **NoteX** reference that has an identifier **n.** (Although you could even say **Play(NoteX NoteX)** without an error.) Thus it appears that the programmer intended to override **Play( )** but mistyped the method a bit. The compiler, however, assumed that an overload and not an override was intended. Note that if you follow the standard C# naming convention, the argument identifier would be **noteX** (lowercase 'n'), which would distinguish it from the class name.

In **Tune**, the **InstrumentX i** is sent the **Play( )** message, with one of **NoteX**'s members (**MIDDLE_C**) as an argument. Since **NoteX** contains **int** definitions, this means that the **int** version of the now-overloaded **Play( )** method is called, and since that has *not* been overridden the base-class version is used.

The output is:

```
InstrumentX.Play()
```

This certainly doesn't appear to be a polymorphic method call. Once you understand what's happening, you can fix the problem fairly easily, but imagine how difficult it might be to find the bug if it's buried in a program of significant size.

Operator Overloading

Abstract classes and methods

In all the instrument examples, the methods in the base class **Instrument** were always "dummy" methods. If these methods are ever called, you've done something wrong. That's because the intent of

Instrument is to create a *common interface* for all the classes derived from it.

The only reason to establish this common interface is so it can be expressed differently for each different subtype. It establishes a basic form, so you can say what's in common with all the derived classes. Another way of saying this is to call **Instrument** an *abstract base class* (or simply an *abstract class*). You create an abstract class when you want to manipulate a set of classes through this common interface. All derived-class methods that match the signature of the base-class declaration will be called using the dynamic binding mechanism. (However, as seen in the last section, if the method's name is the same as the base class but the arguments are different, you've got overloading, which probably isn't what you want.)

If you have an abstract class like **Instrument**, objects of that class almost always have no meaning. That is, **Instrument** is meant to express only the interface, and not a particular implementation, so creating an **Instrument** object makes no sense, and you'll probably want to prevent the user from doing it. This can be accomplished by making all the methods in **Instrument** print error messages, but that delays the information until run-time and requires reliable exhaustive testing on the user's part. It's always better to catch problems at compile-time.

C# provides a mechanism for doing this called the *abstract method*[3]. This is a method that is incomplete; it has only a declaration and no method body. Here is the syntax for an abstract method declaration:

```
abstract void F();
```

A class containing abstract methods is called an *abstract class*. If a class contains one or more abstract methods, the class must be qualified as **abstract**. (Otherwise, the compiler gives you an error message.)

There's no need to qualify **abstract** methods as **virtual**, as they are always resolved with late binding.

---

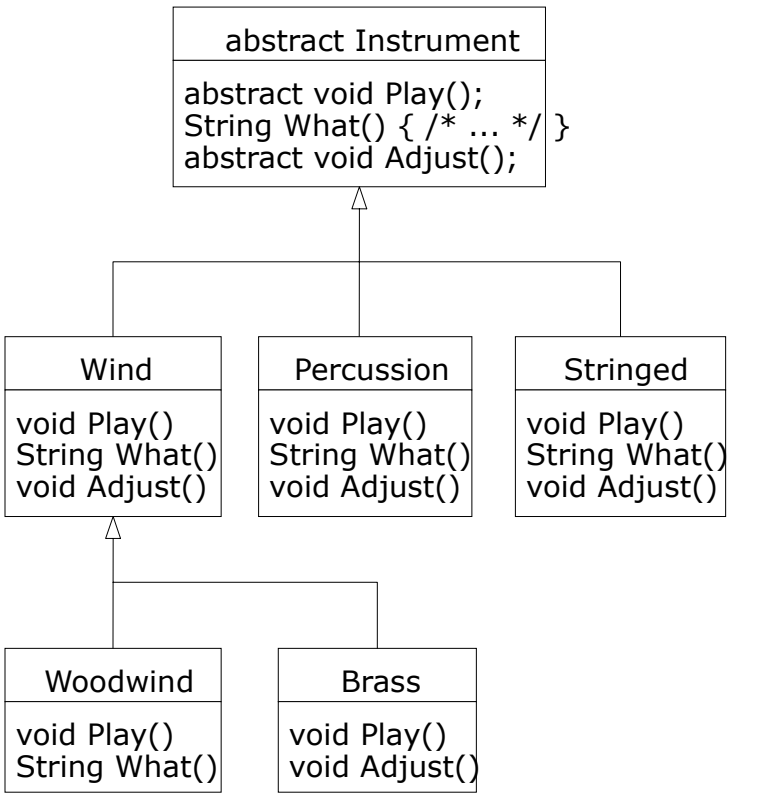
[3] For C++ programmers, this is the analogue of C++'s *pure virtual function*.

If an abstract class is incomplete, what is the compiler supposed to do when someone tries to instantiate an object of that class? It cannot safely create an object of an abstract class, so you get an error message from the compiler. This way the compiler ensures the purity of the abstract class, and you don't need to worry about misusing it.

If you inherit from an abstract class and you want to make objects of the new type, you must provide method definitions for all the abstract methods in the base class. If you don't (and you may choose not to), then the derived class is also abstract and the compiler will force you to qualify *that* class with the **abstract** keyword.

It's possible to create a class as **abstract** without including any **abstract** methods. This is useful when you've got a class in which it doesn't make sense to have any **abstract** methods, and yet you want to prevent any instances of that class.

The **Instrument** class can easily be turned into an **abstract** class. Only some of the methods will be **abstract**, since making a class abstract doesn't force you to make all the methods **abstract**. Here's what it looks like:



Here's the orchestra example modified to use **abstract** classes and methods:

```
//:c07:Music4.cs
// An extensible program.

abstract class Instrument {
  public abstract void Play();

  public virtual string What() {
    return "Instrument";
  }
  public abstract void Adjust();
}
```

```
class Wind : Instrument {
  public override void Play() {
    System.Console.WriteLine("Wind.Play()");
  }
  public override string What() { return "Wind"; }
  public override void Adjust() {}
}

class Percussion : Instrument {
  public override void Play() {
    System.Console.WriteLine("Percussion.Play()");
  }
  public override string What() { return "Percussion";
}
  public override void Adjust() {}
}

class Stringed : Instrument {
  public override void Play() {
    System.Console.WriteLine("stringed.Play()");
  }
  public override string What() { return "stringed"; }
  public override void Adjust() {}
}

class Brass : Wind {
  public override void Play() {
    System.Console.WriteLine("Brass.Play()");
  }
  public override void Adjust() {
    System.Console.WriteLine("Brass.Adjust()");
  }
}
```

```
class Woodwind : Wind {
  public override void Play() {
    System.Console.WriteLine("Woodwind.Play()");
  }
  public override string What() { return "Woodwind"; }
}

public class Music3 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void Tune(Instrument i) {
    // ...
    i.Play();
  }
  static void TuneAll(Instrument[] e) {
    foreach(Instrument i in e)
      Tune(i);
  }
  public static void Main() {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    TuneAll(orchestra);
  }
} ///:~
```

You can see that there's really no change except in the base class.

It's helpful to create **abstract** classes and methods because they make the abstractness of a class explicit, and tell both the user and the compiler how it was intended to be used.

Constructors and polymorphism

As usual, constructors are different from other kinds of methods. This is also true when polymorphism is involved. Even though constructors are not polymorphic (although you can have a kind of "virtual constructor," as you will see in Chapter #virtual constructor#), it's important to understand the way constructors work in complex hierarchies and with polymorphism. This understanding will help you avoid unpleasant entanglements.

Order of constructor calls

The order of constructor calls was briefly discussed in Chapter #initialization#, but that was before polymorphism was introduced.

A constructor for the base class is always called in the constructor for a derived class, chaining up the inheritance hierarchy so that a constructor for every base class is called. This makes sense because the constructor has a special job: to see that the object is built properly. A derived class has access to its own members only, and not to **private** members of the base class. Only the base-class constructor has the proper knowledge and access to initialize its own elements. Therefore, it's essential that all constructors get called, otherwise the entire object wouldn't be consistently constructed. That's why the compiler enforces a constructor call for every portion of a derived class. It will silently call the default constructor if you don't explicitly call a base-class constructor in the derived-class constructor body. If there is no default constructor, the compiler will complain. (In the case where a class has no constructors, the compiler will automatically synthesize a default constructor.)

Let's take a look at an example that shows the effects of composition, inheritance, and polymorphism on the order of construction:

```
//:c07:Sandwich.cs
// Order of constructor calls.

public class Meal {
  internal Meal() {
System.Console.WriteLine("Meal()"); }
}

public class Bread {
  internal Bread() {
System.Console.WriteLine("Bread()"); }
}

public class Cheese {
  internal Cheese() {
System.Console.WriteLine("Cheese()"); }
}

public class Lettuce {
  internal Lettuce() {
System.Console.WriteLine("Lettuce()"); }
}

public class Lunch : Meal {
  internal Lunch() {
System.Console.WriteLine("Lunch()");}
}

public class PortableLunch : Lunch {
  internal PortableLunch() {
    System.Console.WriteLine("PortableLunch()");
  }
}

public class Sandwich : PortableLunch {
  Bread b = new Bread();
  Cheese c = new Cheese();
  Lettuce l = new Lettuce();
  internal Sandwich() {
    System.Console.WriteLine("Sandwich()");
  }
```

```
  public static void Main() {
    new Sandwich();
  }
} ///:~
```

This example creates a complex class out of other classes, and each class has a constructor that announces itself. The important class is **Sandwich**, which reflects three levels of inheritance (four, if you count the implicit inheritance from **object**) and three member objects. When a **Sandwich** object is created in **Main( )**, the output is:

```
Bread()
Cheese()
Lettuce()
Meal()
Lunch()
PortableLunch()
Sandwich()
```

This means that the order of constructor calls for a complex object is as follows:

4. Member initializers are called in the order of declaration
5. The base-class constructor is called. This step is repeated recursively such that the root of the hierarchy is constructed first, followed by the next-derived class, etc., until the most-derived class is reached.
6. The body of the derived-class constructor is called.

The order of the constructor calls is important. When you inherit, you know all about the base class and can access any **public**, **protected,** or **internal** members of the base class. This means that you must be able to assume that all the members of the base class are valid when you're in the derived class. In a normal method, construction has already taken place, so all the members of all parts of the object have been built. Inside the constructor, however, you must be able to assume that all members that you use have been built. The only way to guarantee this is for the base-class constructor to be called first. Then when you're in the derived-class constructor, all the members you can access in the base class have been initialized. "Knowing that all members are valid" inside the constructor is also the reason that, whenever possible, you should initialize all member objects (that is, objects placed in the class using composition) at their

point of definition in the class (e.g., **b**, **c,** and **l** in the example above). If you follow this practice, you will help ensure that all base class members *and* member objects of the current object have been initialized. Unfortunately, this doesn't handle every case, as you will see in the next section.

Behavior of polymorphic methods inside constructors

The hierarchy of constructor calls brings up an interesting dilemma. What happens if you're inside a constructor and you call a dynamically bound method of the object being constructed? Inside an ordinary method you can imagine what will happen—the dynamically bound call is resolved at run-time because the object cannot know whether it belongs to the class that the method is in or some class derived from it. For consistency, you might think this is what should happen inside constructors.

This is not exactly the case. If you call a dynamically bound method inside a constructor, the overridden definition for that method is used. However, the *effect* can be rather unexpected, and can conceal some difficult-to-find bugs.

Conceptually, the constructor's job is to bring the object into existence (which is hardly an ordinary feat). Inside any constructor, the entire object might be only partially formed—you can know only that the base-class objects have been initialized, but you cannot know which classes are inherited from you. A dynamically bound method call, however, reaches "outward" into the inheritance hierarchy. It calls a method in a derived class. If you do this inside a constructor, you call a method that might manipulate members that haven't been initialized yet—a sure recipe for disaster.

You can see the problem in the following example:

```
//:c07:PolyConstructors.cs
// Constructors and polymorphism
// don't produce what you might expect.

abstract class Glyph {
```

```
  protected abstract void Draw();
  internal Glyph() {
    System.Console.WriteLine("Glyph() before draw()");
    Draw();
    System.Console.WriteLine("Glyph() after draw()");
  }
}

class RoundGlyph : Glyph {
  int radius = 1;
  int thickness;
  internal RoundGlyph(int r) {
    radius = r;
    thickness = 2;
    System.Console.WriteLine(
      "RoundGlyph.RoundGlyph(), radius = {0} thickness
= {1}",
      radius, thickness);
  }
  protected override void Draw() {
    System.Console.WriteLine(
      "RoundGlyph.Draw(), radius = {0} thickness =
{1}",
       radius, thickness);
  }
}

public class PolyConstructors {
  public static void Main() {
    new RoundGlyph(5);
  }
} ///:~
```

In **Glyph**, the **Draw( )** method is **abstract**, so it is designed to be overridden. Indeed, you are forced to override it in **RoundGlyph**. But

the **Glyph** constructor calls this method, and the call ends up in **RoundGlyph.Draw( )**, which would seem to be the intent. But look at the output:

```
Glyph() before draw()
RoundGlyph.Draw(), radius = 1 thickness = 0
Glyph() after draw()
RoundGlyph.RoundGlyph(), radius = 5 thickness = 2
```

When **Glyph**'s constructor calls **Draw( )**, the value of **radius** are set to their default values, not their post-construction intended values.

A good guideline for constructors is, "If possible, initialize member variables directly. Do as little as possible in a constructor to set the object into a good state, and if you can possibly avoid it, don't call any methods." The only safe methods to call inside a constructor are non-virtual.

Designing with inheritance

Once you learn about polymorphism, it can seem that everything ought to be inherited because polymorphism is such a clever tool. This can burden your designs; in fact if you choose inheritance first when you're using an existing class to make a new class, things can become needlessly complicated.

A better approach is to choose composition first, when it's not obvious which one you should use. Composition does not force a design into an inheritance hierarchy. But composition is also more flexible since it's possible to dynamically choose a type (and thus behavior) when using composition, whereas inheritance requires an exact type to be known at compile-time. The following example illustrates this:

```
//:c07:Transmogrify.cs
// Dynamically changing the behavior of
// an object via composition.

abstract class Actor {
  public abstract void Act();
}
```

```
class HappyActor : Actor {
  public override void Act() {
    System.Console.WriteLine("HappyActor");
  }
}

class SadActor : Actor {
  public override void Act() {
    System.Console.WriteLine("SadActor");
  }
}

class Stage {
  Actor a = new HappyActor();
  internal void Change() { a = new SadActor(); }
  internal void Go() { a.Act(); }
}

public class Transmogrify {
  public static void Main() {
    Stage s = new Stage();
    s.Go(); // Prints "HappyActor"
    s.Change();
    s.Go(); // Prints "SadActor"
  }
} ///:~
```

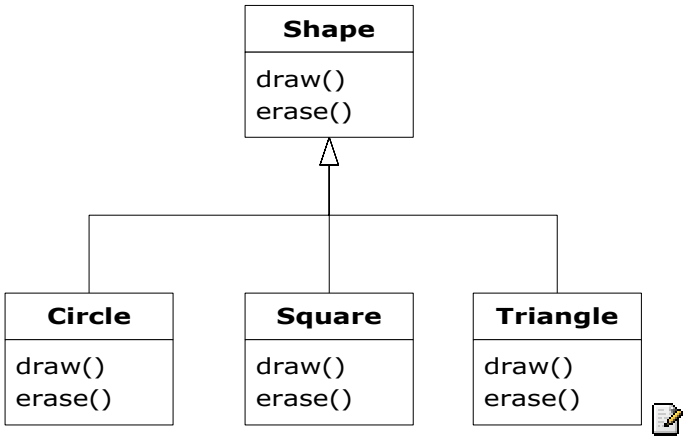
A **Stage** object contains a reference to an **Actor**, which is initialized to a **HappyActor** object. This means **Go( )** produces a particular behavior. But since a reference can be rebound to a different object at run-time, a reference for a **SadActor** object can be substituted in **a** and then the behavior produced by **Go( )** changes. Thus you gain dynamic flexibility at run-time. (This is also called the *State Pattern*. See *Thinking in Patterns*, downloadable at *www.BruceEckel.com*.) In contrast, you can't decide to inherit differently at run-time; that must be completely determined at compile-time.

A general guideline is "Use inheritance to express differences in behavior, and fields to express variations in state." In the above example, both are used: two different classes are inherited to express the difference in the

Act( ) method, and **Stage** uses composition to allow its state to be changed. In this case, that change in state happens to produce a change in behavior.

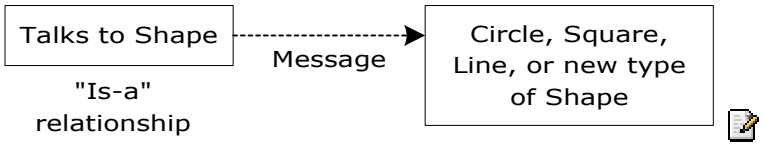
Pure inheritance vs. extension

When studying inheritance, it would seem that the cleanest way to create an inheritance hierarchy is to take the "pure" approach. That is, only methods that have been established in the base class or **interface** are to be overridden in the derived class, as seen in this diagram:



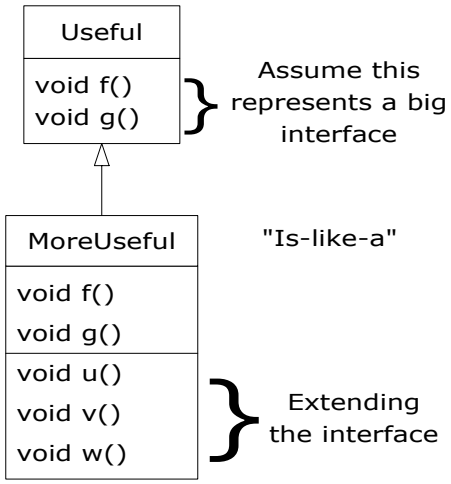
This can be termed a pure "is-a" relationship because the interface of a class establishes what it is. Inheritance guarantees that any derived class will have the interface of the base class and nothing less. If you follow the above diagram, derived classes will also have *no more* than the base class interface.

This can be thought of as *pure substitution*, because derived class objects can be perfectly substituted for the base class, and you never need to know any extra information about the subclasses when you're using them:

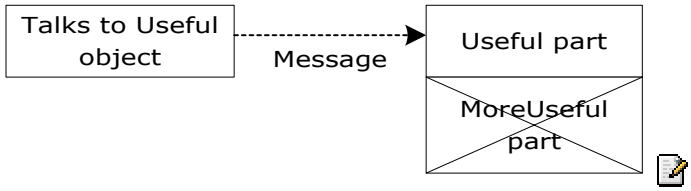


That is, the base class can receive any message you can send to the derived class because the two have exactly the same interface. All you need to do is upcast from the derived class and never look back to see what exact type of object you're dealing with. Everything is handled through polymorphism.

When you see it this way, it seems like a pure "is-a" relationship is the only sensible way to do things, and any other design indicates muddled thinking and is by definition broken. This too is a trap. As soon as you start thinking this way, you'll turn around and discover that extending the interface is the perfect solution to a particular problem. This could be termed an "is-like-a" relationship because the derived class is *like* the base class—it has the same fundamental interface—but it has other features that require additional methods to implement:



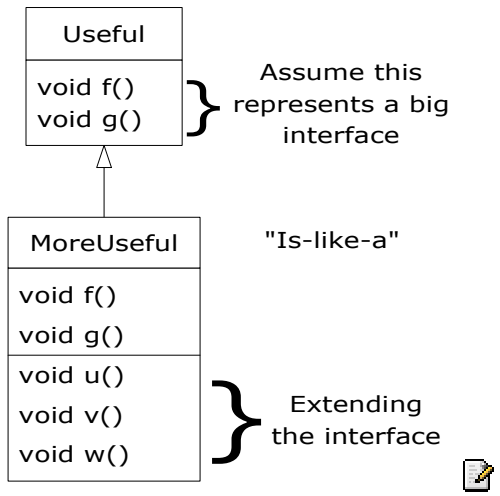
While this is also a useful and sensible approach (depending on the situation) it has a drawback. The extended part of the interface in the derived class is not available from the base class, so once you upcast you can't call the new methods:



If you're not upcasting in this case, it won't bother you, but often you'll get into a situation in which you need to rediscover the exact type of the object so you can access the extended methods of that type. The following section shows how this is done.

Downcasting and run-time type identification

Since you lose the specific type information via an *upcast* (moving up the inheritance hierarchy), it makes sense that to retrieve the type information—that is, to move back down the inheritance hierarchy—you use a *downcast*. However, you know an upcast is always safe; the base class cannot have a bigger interface than the derived class, therefore every message you send through the base class interface is guaranteed to be accepted. But with a downcast, you don't really know that a shape (for example) is actually a circle. It could instead be a triangle or square or some other type.



To solve this problem there must be some way to guarantee that a downcast is correct, so you won't accidentally cast to the wrong type and then send a message that the object can't accept. This would be quite unsafe. In some languages (like C++) you must perform a special operation in order to get a type-safe downcast, but in C# *every cast* is checked!

C# supports two types of downcast: a parenthesized cast that looks similar to the casts in other C-derived languages:

```
MoreUseful downCastObject = (MoreUseful)
myUsefulHandle;
```

and the **as** keyword:

```
MoreUseful downCastObject = myUsefulHandle as
MoreUseful;
```

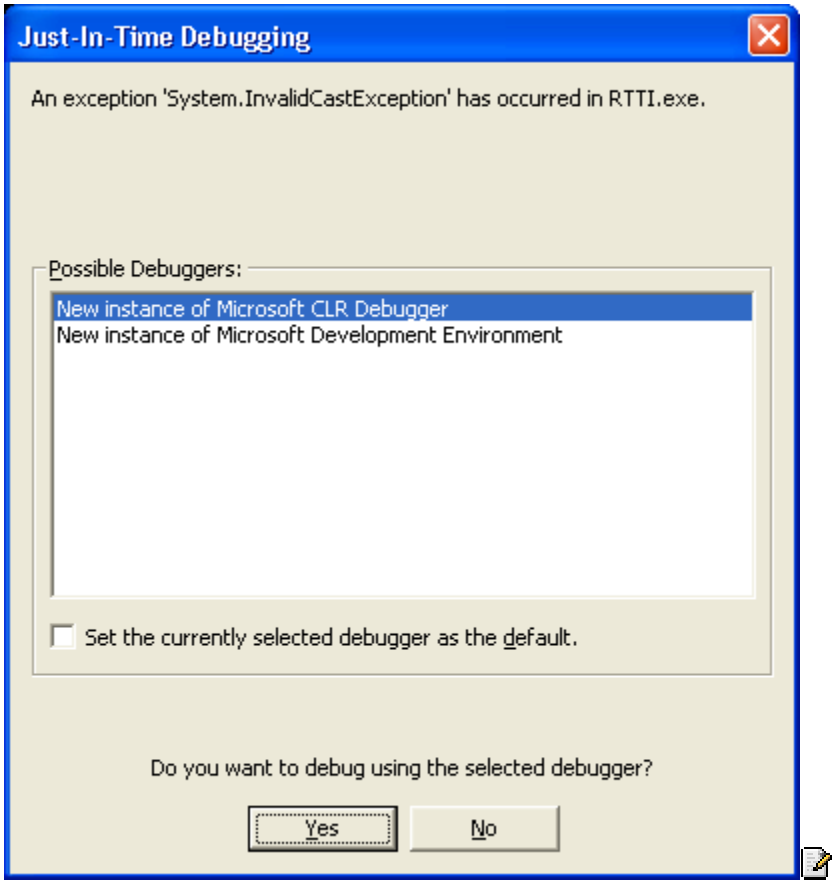
At run-time, both these casts are checked to ensure that the **myUsefulHandle** does in fact refer to an instance of type **MoreUseful.** If this a bad assumption, the parenthesize cast will throw an **InvalidCastException** and the **as** cast will assign **downCastObject** the value of null.

This act of checking types at run-time is called *run-time type identification* (RTTI). The following example demonstrates the behavior of RTTI:

```
//:c07:RTTI.CS
// Downcasting & Run-time Type
// Identification (RTTI).

class Useful {
  public virtual void F() {}
  public virtual void G() {}
}

class MoreUseful : Useful {
  public override void F() {}
  public override void G() {}
```

```
  public void U() {}
  public void V() {}
  public void W() {}
}

public class RTTI {
  public static void Main() {
    Useful[] x = {
      new Useful(),
      new MoreUseful()
    };
    x[0].F();
    x[1].G();
    // Compile-time: method not found in Useful:
    //! x[1].U();
    ((MoreUseful)x[1]).U(); // Parenthesized downcast
    (x[1] as MoreUseful).U(); //as keyword
    ((MoreUseful)x[0]).U(); // Exception thrown
  }
} ///:~
```

When you run this program, you will see something we've not yet discussed, Visual Studio's Just-In-Time Debugging dialogue:


Just-In-Time Debugging
An exception 'System.InvalidCastException' has occurred in RTTI.exe.
Possible Debuggers:
New instance of Microsoft CLR Debugger
New instance of Microsoft Development Environment
Set the currently selected debugger as the default.
Do you want to debug using the selected debugger?
Yes
No

This is certainly more welcome than a Dr. Watson dump or a Blue Screen of Death, but we know the cause – we're trying to treat **x[0]** as a **MoreUseful** when it's only a **Useful**. Select "No" and the program will end with a complaint about an unhandled **InvalidCastException**.

As in the diagram, **MoreUseful** extends the interface of **Useful**. But since it's inherited, it can also be upcast to a **Useful**. You can see this happening in the initialization of the array **x** in **Main( )**. Since both objects in the array are of class **Useful**, you can send the **F( )** and **G( )** methods to both, and if you try to call **U( )** (which exists only in **MoreUseful**) you'll get a compile-time error message.

If you want to access the extended interface of a **MoreUseful** object, you can try to downcast. If it's the correct type, it will be successful. Otherwise, you'll get an **InvalidCastException**.

There's more to RTTI than a simple cast. The **is** keyword allows you check the type of an object before attempting a downcast.

```
//:c07:RTTI2.cs
// Downcasting & Run-time Type
// Identification (RTTI).

class Useful {
  public virtual void F() {}
  public virtual void G() {}
}

class MoreUseful : Useful {
  public override void F() {}
  public override void G() {}
  public void U() {}
  public void V() {}
  public void W() {}
}

public class RTTI2 {
  public static void Main() {
    Useful[] x = {
      new Useful(),
      new MoreUseful()
    };
    x[0].F();
    x[1].G();

    foreach(Useful u in x){
        if(u is MoreUseful){
            ((MoreUseful) u).U();
        }
    }
  }
} ///:~
```

This program runs to completion without any exceptions being thrown. Everything stays the same except the final iteration over the **x** array. The **foreach** loop iterates over the array (all two elements of it!), but we guard the downcast with a Boolean test to ensure that we only attempt the downcast on objects of type **MoreUseful**.

Summary

Polymorphism means "different forms." In object-oriented programming, you have the same face (the common interface in the base class) and different forms using that face: the different versions of the dynamically bound methods.

You've seen in this chapter that it's impossible to understand, or even create, an example of polymorphism without using data abstraction and inheritance. Polymorphism is a feature that cannot be viewed in isolation (like a **switch** statement can, for example), but instead works only in concert, as part of a "big picture" of class relationships. People are often confused by other, non-object-oriented features of C#, like method overloading, which are sometimes presented as object-oriented. Don't be fooled: If it isn't late binding, it isn't polymorphism.

To use polymorphism—and thus object-oriented techniques—effectively in your programs you must expand your view of programming to include not just members and messages of an individual class, but also the commonality among classes and their relationships with each other. Although this requires significant effort, it's a worthy struggle, because the results are faster program development, better code organization, extensible programs, and easier code maintenance.

Exercises

8a: Interfaces

Interfaces provide more sophisticated ways to organize and control the objects in your system.

C++, for example, does not contain such mechanisms, although the clever programmer may simulate them. The fact that they exist in Java indicates that they were considered important enough to provide direct support through language keywords.

In Chapter 7, you learned about the **abstract** keyword, which allows you to create one or more methods in a class that have no definitions—you provide part of the interface without providing a corresponding implementation, which is created by inheritors. The **interface** keyword produces a completely abstract class, one that provides no implementation at all. You'll learn that the **interface** is more than just an abstract class taken to the extreme, since it allows you to perform a variation on C++'s "multiple inheritance," by creating a class that can be upcast to more than one base type.

At first, inner classes look like a simple code-hiding mechanism: you place classes inside other classes. You'll learn, however, that the inner class does more than that—it knows about and can communicate with the surrounding class—and that the kind of code you can write with inner classes is more elegant and clear, although it is a new concept to most. It takes some time to become comfortable with design using inner classes.

Interfaces

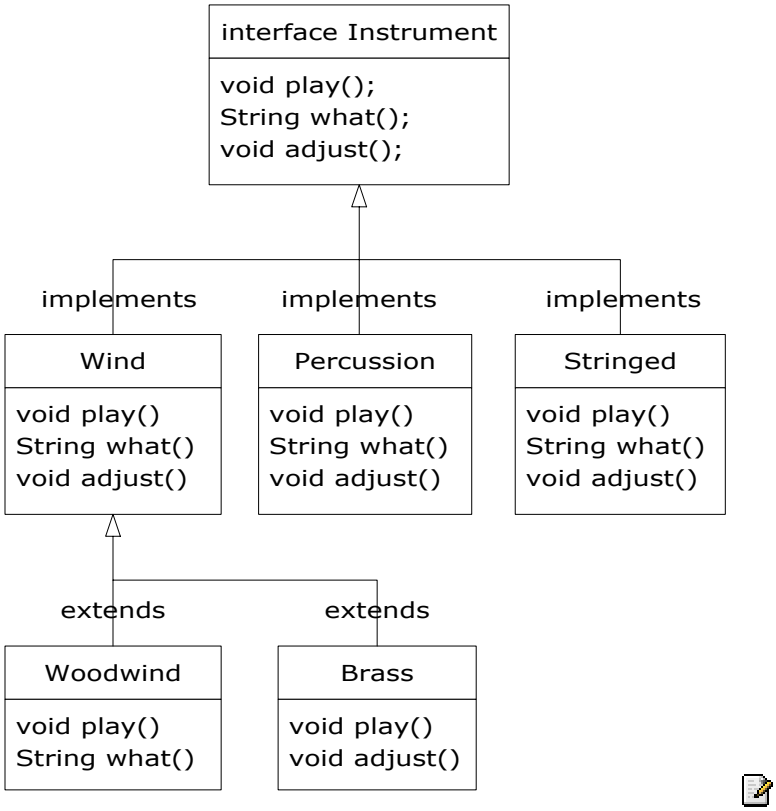
The **interface** keyword takes the **abstract** concept one step further. You could think of it as a "pure" **abstract** class. It allows the creator to establish the form for a class: method names, argument lists, and return types, but no method bodies. An **interface** can also contain fields, but

these are implicitly **static** and **final**. An **interface** provides only a form, but no implementation.

An **interface** says: "This is what all classes that *implement* this particular interface will look like." Thus, any code that uses a particular **interface** knows what methods might be called for that **interface**, and that's all. So the **interface** is used to establish a "protocol" between classes. (Some object-oriented programming languages have a keyword called *protocol* to do the same thing.)

To create an **interface**, use the **interface** keyword instead of the **class** keyword. Like a class, you can add the **public** keyword before the **interface** keyword (but only if that **interface** is defined in a file of the same name) or leave it off to give "friendly" status so that it is only usable within the same package.

To make a class that conforms to a particular **interface** (or group of **interface**s) use the **implements** keyword. You're saying "The **interface** is what it looks like but now I'm going to say how it *works*." Other than that, it looks like inheritance. The diagram for the instrument example shows this:



Once you've implemented an **interface**, that implementation becomes an ordinary class that can be extended in the regular way.

You can choose to explicitly declare the method declarations in an **interface** as **public**. But they are **public** even if you don't say it. So when you **implement** an **interface**, the methods from the **interface** must be defined as **public**. Otherwise they would default to "friendly," and you'd be reducing the accessibility of a method during inheritance, which is not allowed by the Java compiler.

You can see this in the modified version of the **Instrument** example. Note that every method in the **interface** is strictly a declaration, which is the only thing the compiler allows. In addition, none of the methods in **Instrument** are declared as **public**, but they're automatically **public** anyway:

N.B.: Implementing interface *can't* be declared virtual, since it's not redefining a class. Note that interfaces can't have compile-time constants (check).

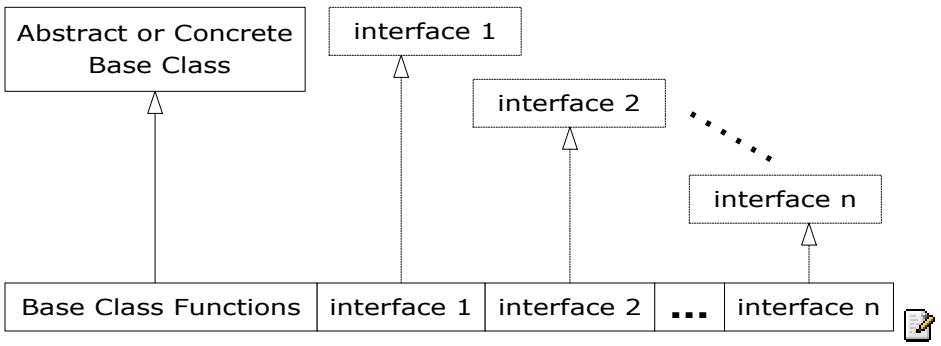
```
//:c08:Music5.cs
// Interfaces.

interface Instrument {
  // Compile-time constant:
  //! No compile constants int i = 5;
  // Cannot have method definitions:
  void Play(); // Automatically public
  string What();
  void Adjust();
}

class Wind : Instrument {
  public virtual void Play() {
    System.Console.WriteLine("Wind.Play()");
  }
  public virtual string What() { return "Wind"; }
  public virtual void Adjust() {}
}

class Percussion : Instrument {
  public virtual void Play() {
    System.Console.WriteLine("Percussion.Play()");
  }
  public virtual string What() { return "Percussion";
}
  public virtual void Adjust() {}
}

class Stringed : Instrument {
  public virtual void Play() {
    System.Console.WriteLine("stringed.Play()");
  }
  public virtual string What() { return "stringed"; }
  public virtual void Adjust() {}
}
```

```
class Brass : Wind {
  public override void Play() {
    System.Console.WriteLine("Brass.Play()");
  }
  public override void Adjust() {
    System.Console.WriteLine("Brass.Adjust()");
  }
}

class Woodwind : Wind {
  public override void Play() {
    System.Console.WriteLine("Woodwind.Play()");
  }
  public override string What() { return "Woodwind"; }
}

public class Music5 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument i) {
    // ...
    i.Play();
  }
  static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.Length; i++)
      tune(e[i]);
  }
  public static void Main() {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    tuneAll(orchestra);
  }
} ///:~
```

The rest of the code works the same. It doesn't matter if you are upcasting to a "regular" class called **Instrument**, an **abstract** class called **Instrument**, or to an **interface** called **Instrument**. The behavior is the same. In fact, you can see in the **tune( )** method that there isn't any evidence about whether **Instrument** is a "regular" class, an **abstract** class, or an **interface**. This is the intent: Each approach gives the programmer different control over the way objects are created and used.

"Multiple inheritance" in Java

The **interface** isn't simply a "more pure" form of **abstract** class. It has a higher purpose than that. Because an **interface** has no implementation at all—that is, there is no storage associated with an **interface**—there's nothing to prevent many **interface**s from being combined. This is valuable because there are times when you need to say "An **x** is an **a** *and* a **b** *and* a **c**." In C++, this act of combining multiple class interfaces is called *multiple inheritance*, and it carries some rather sticky baggage because each class can have an implementation. In Java, you can perform the same act, but only one of the classes can have an implementation, so the problems seen in C++ do not occur with Java when combining multiple interfaces:



In a derived class, you aren't forced to have a base class that is either an **abstract** or "concrete" (one with no **abstract** methods). If you *do* inherit from a non-**interface**, you can inherit from only one. All the rest of the base elements must be **interface**s. You place all the interface names after the **implements** keyword and separate them with commas. You can have as many **interface**s as you want—each one becomes an independent type

that you can upcast to. The following example shows a concrete class combined with several **interface**s to produce a new class:

```
//:c08:Adventure.cs
// Multiple interfaces.

interface ICanFight {
  void Fight();
}

interface ICanSwim {
  void Swim();
}

interface ICanFly {
  void Fly();
}

class ActionCharacter {
  public void Fight()
{System.Console.WriteLine("Fighting");}
}

class Hero : ActionCharacter, ICanFight, ICanSwim,
ICanFly {
  public void Swim()
{System.Console.WriteLine("Swimming");}
  public void Fly()
{System.Console.WriteLine("Flying");}
}

public class Adventure {
  static void T(ICanFight x) { x.Fight(); }
  static void U(ICanSwim x) { x.Swim(); }
  static void V(ICanFly x) { x.Fly(); }
  static void W(ActionCharacter x) { x.Fight(); }
  public static void Main() {
    Hero h = new Hero();
    T(h); // Treat it as an ICanFight
    U(h); // Treat it as an ICanSwim
    V(h); // Treat it as an ICanFly
```

```
    W(h); // Treat it as an ActionCharacter
  }
} ///:~
```

You can see that **Hero** combines the concrete class **ActionCharacter** with the interfaces **CanFight**, **CanSwim**, and **CanFly**. When you combine a concrete class with interfaces this way, the concrete class must come first, then the interfaces. (The compiler gives an error otherwise.)

Note that the signature for **fight( )** is the same in the **interface CanFight** and the class **ActionCharacter**, and that **fight( )** is *not* provided with a definition in **Hero**. The rule for an **interface** is that you can inherit from it (as you will see shortly), but then you've got another **interface**. If you want to create an object of the new type, it must be a class with all definitions provided. Even though **Hero** does not explicitly provide a definition for **fight( )**, the definition comes along with **ActionCharacter** so it is automatically provided and it's possible to create objects of **Hero**.

In class **Adventure**, you can see that there are four methods that take as arguments the various interfaces and the concrete class. When a **Hero** object is created, it can be passed to any of these methods, which means it is being upcast to each **interface** in turn. Because of the way interfaces are designed in Java, this works without a hitch and without any particular effort on the part of the programmer.

Keep in mind that the core reason for interfaces is shown in the above example: to be able to upcast to more than one base type. However, a second reason for using interfaces is the same as using an **abstract** base class: to prevent the client programmer from making an object of this class and to establish that it is only an interface. This brings up a question: Should you use an **interface** or an **abstract** class? An **interface** gives you the benefits of an **abstract** class *and* the benefits of an **interface**, so if it's possible to create your base class without any method definitions or member variables you should always prefer **interface**s to **abstract** classes. In fact, if you know something is going to be a base class, your first choice should be to make it an **interface**, and only if you're forced to have method definitions or member variables should you change to an **abstract** class, or if necessary a concrete class.

Name collisions when combining interfaces

You can encounter a small pitfall when implementing multiple interfaces. In the above example, both **CanFight** and **ActionCharacter** have an identical **void fight( )** method. This is no problem because the method is identical in both cases, but what if it's not? Here's an example: Explicit interface implementation

```
//:c08:InterfaceCollision.cs
interface I1 { void F(); }
interface I2 { int F(int i); }
interface I3 { int F(); }
class C { public virtual int F() { return 1; } }

class C2 : I1, I2 {
  public void F() {}
  public int F(int i) { return 1; }
}

class C3 : C, I2 {
  public int F(int i) { return 1; }
}

class C4 : C, I3 {
  // Identical, no problem:
  public override int F() { return 1; }
}

class C5 : C , I1 {
    public override int F(){ return 1; }
    void I1.F(){  }
}

interface I4 : I1, I3 {}

class C6: I4{
   void I1.F() { }
   int I3.F() { return 1; }
}///:~
```

The difficulty occurs because overriding, implementation, and overloading get unpleasantly mixed together, and overloaded functions cannot differ only by return type. When the last two lines are uncommented, the error messages say it all:

InterfaceCollision.java:23: f() in C cannot implement f() in I1; attempting to use incompatible return type
found : int
required: void
InterfaceCollision.java:24: interfaces I3 and I1 are incompatible; both define f(), but with different return type

Using the same method names in different interfaces that are intended to be combined generally causes confusion in the readability of the code, as well. Strive to avoid it.

Extending an interface with inheritance

You can easily add new method declarations to an **interface** using inheritance, and you can also combine several **interface**s into a new **interface** with inheritance. In both cases you get a new **interface**, as seen in this example:

```
//:c08:HorrorShow.cs
// Extending an interface with inheritance.

interface Monster {
  void Menace();
}

interface DangerousMonster : Monster {
  void Destroy();
}

interface Lethal {
  void Kill();
}

class DragonZilla : DangerousMonster {
```

```
  public void Menace() {}
  public void Destroy() {}
}

interface Vampire  : DangerousMonster, Lethal {
  void DrinkBlood();
}

public class HorrorShow {
  static void U(Monster b) { b.Menace(); }
  static void V(DangerousMonster d) {
    d.Menace();
    d.Destroy();
  }
  public static void Main() {
    DragonZilla if2 = new DragonZilla();
    U(if2);
    V(if2);
  }
} ///:~
```

DangerousMonster is a simple extension to **Monster** that produces a new **interface**. This is implemented in **DragonZilla**.

The syntax used in **Vampire** works *only* when inheriting interfaces. Normally, you can use **extends** with only a single class, but since an **interface** can be made from multiple other interfaces, **extends** can refer to multiple base interfaces when building a new **interface**. As you can see, the **interface** names are simply separated with commas.

Doesn't work in C#. Must have section on enums and structs earlier

Because any fields you put into an **interface** are automatically **static** and **final**, the **interface** is a convenient tool for creating groups of constant values, much as you would with an **enum** in C or C++. For example:

```
//:c08:Months.java
// Using interfaces to create groups of constants.
```

```
package c08;

public interface Months {
  int
    JANUARY = 1, FEBRUARY = 2, MARCH = 3,
    APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
    AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
    NOVEMBER = 11, DECEMBER = 12;
} ///:~
```

Notice the Java style of using all uppercase letters (with underscores to separate multiple words in a single identifier) for **static final**s that have constant initializers.

The fields in an **interface** are automatically **public**, so it's unnecessary to specify that.

Now you can use the constants from outside the package by importing **c08.*** or **c08.Months** just as you would with any other package, and referencing the values with expressions like **Months.JANUARY**. Of course, what you get is just an **int**, so there isn't the extra type safety that C++'s **enum** has, but this (commonly used) technique is certainly an improvement over hard-coding numbers into your programs. (That approach is often referred to as using "magic numbers" and it produces very difficult-to-maintain code.)

If you do want extra type safety, you can build a class like this[1]:

```
//: c08:Month2.java
// A more robust enumeration system.
package c08;

public final class Month2 {
  private String name;
  private int order;
  private Month2(int ord, String nm) {
    order = ord;
    name = nm;
```

[1] This approach was inspired by an e-mail from Rich Hoffarth.

```
  }
  public String toString() { return name; }
  public final static Month2
    JAN = new Month2(1, "January"),
    FEB = new Month2(2, "February"),
    MAR = new Month2(3, "March"),
    APR = new Month2(4, "April"),
    MAY = new Month2(5, "May"),
    JUN = new Month2(6, "June"),
    JUL = new Month2(7, "July"),
    AUG = new Month2(8, "August"),
    SEP = new Month2(9, "September"),
    OCT = new Month2(10, "October"),
    NOV = new Month2(11, "November"),
    DEC = new Month2(12, "December");
  public final static Month2[] month =  {
    JAN, FEB, MAR, APR, MAY, JUN,
    JUL, AUG, SEP, OCT, NOV, DEC
  };
  public final static Month2 number(int ord) {
    return month[ord - 1];
  }
  public static void main(String[] args) {
    Month2 m = Month2.JAN;
    System.out.println(m);
    m = Month2.number(12);
    System.out.println(m);
    System.out.println(m == Month2.DEC);
    System.out.println(m.equals(Month2.DEC));
  }
} ///:~
```

The class is called **Month2**, since there's already a **Month** in the standard Java library. It's a **final** class with a **private** constructor so no one can inherit from it or make any instances of it. The only instances are the **final static** ones created in the class itself: **JAN**, **FEB**, **MAR**, etc. These objects are also used in the array **month**, which lets you iterate through an array of **Month2** objects. The **number( )** method allows you to select a **Month2** by giving its corresponding month number. In **main( )** you can see the type safety: **m** is a **Month2** object so it can be assigned only to a **Month2**. The previous example **Months.java**

provided only **int** values, so an **int** variable intended to represent a month could actually be given any integer value, which wasn't very safe.

This approach also allows you to use == or **equals( )** interchangeably, as shown at the end of **main( )**. This works because there can be only one instance of each value of **Month2**.

Initializing fields in interfaces

Fields defined in interfaces are automatically **static** and **final**. These cannot be "blank finals," but they can be initialized with nonconstant expressions. For example:

```
//: c08:RandVals.java
// Initializing interface fields with
// non-constant initializers.
import java.util.*;

public interface RandVals {
  int rint = (int)(Math.random() * 10);
  long rlong = (long)(Math.random() * 10);
  float rfloat = (float)(Math.random() * 10);
  double rdouble = Math.random() * 10;
} ///:~
```

Since the fields are **static**, they are initialized when the class is first loaded, which happens when any of the fields are accessed for the first time. Here's a simple test:

```
//: c08:TestRandVals.java

public class TestRandVals {
  public static void main(String[] args) {
    System.out.println(RandVals.rint);
    System.out.println(RandVals.rlong);
    System.out.println(RandVals.rfloat);
    System.out.println(RandVals.rdouble);
  }
} ///:~
```

The fields, of course, are not part of the interface but instead are stored in the **static** storage area for that interface.

Nesting interfaces

[2]Interfaces may be nested within classes and within other interfaces. This reveals a number of very interesting features:

```
//: c08:NestingInterfaces.java

class A {
  interface B {
    void f();
  }
  public class BImp implements B {
    public void f() {}
  }
  private class BImp2 implements B {
    public void f() {}
  }
  public interface C {
    void f();
  }
  class CImp implements C {
    public void f() {}
  }
  private class CImp2 implements C {
    public void f() {}
  }
  private interface D {
    void f();
  }
  private class DImp implements D {
    public void f() {}
  }
  public class DImp2 implements D {
    public void f() {}
  }
```

[2] Thanks to Martin Danner for asking this question during a seminar.

```
  public D getD() { return new DImp2(); }
  private D dRef;
  public void receiveD(D d) {
    dRef = d;
    dRef.f();
  }
}

interface E {
  interface G {
    void f();
  }
  // Redundant "public":
  public interface H {
    void f();
  }
  void g();
  // Cannot be private within an interface:
  //! private interface I {}
}

public class NestingInterfaces {
  public class BImp implements A.B {
    public void f() {}
  }
  class CImp implements A.C {
    public void f() {}
  }
  // Cannot implement a private interface except
  // within that interface's defining class:
  //! class DImp implements A.D {
  //!  public void f() {}
  //! }
  class EImp implements E {
    public void g() {}
  }
  class EGImp implements E.G {
    public void f() {}
  }
  class EImp2 implements E {
    public void g() {}
```

```
    class EG implements E.G {
      public void f() {}
    }
  }
  public static void main(String[] args) {
    A a = new A();
    // Can't access A.D:
    //! A.D ad = a.getD();
    // Doesn't return anything but A.D:
    //! A.DImp2 di2 = a.getD();
    // Cannot access a member of the interface:
    //! a.getD().f();
    // Only another A can do anything with getD():
    A a2 = new A();
    a2.receiveD(a.getD());
  }
} ///:~
```

The syntax for nesting an interface within a class is reasonably obvious, and just like non-nested interfaces these can have **public** or "friendly" visibility. You can also see that both **public** and "friendly" nested interfaces can be implemented as **public**, "friendly," and **private** nested classes.

As a new twist, interfaces can also be **private** as seen in **A.D** (the same qualification syntax is used for nested interfaces as for nested classes). What good is a **private** nested interface? You might guess that it can only be implemented as a **private** nested class as in **DImp**, but **A.DImp2** shows that it can also be implemented as a **public** class. However, **A.DImp2** can only be used as itself. You are not allowed to mention the fact that it implements the **private** interface, so implementing a **private** interface is a way to force the definition of the methods in that interface without adding any type information (that is, without allowing any upcasting).

The method **getD( )** produces a further quandary concerning the **private** interface: it's a **public** method that returns a reference to a **private** interface. What can you do with the return value of this method? In **main( )**, you can see several attempts to use the return value, all of which fail. The only thing that works is if the return value is handed to an object

that has permission to use it—in this case, another **A**, via the **receiveD( )** method.

Interface **E** shows that interfaces can be nested within each other. However, the rules about interfaces—in particular, that all interface elements must be **public**—are strictly enforced here, so an interface nested within another interface is automatically **public** and cannot be made **private**.

NestingInterfaces shows the various ways that nested interfaces can be implemented. In particular, notice that when you implement an interface, you are not required to implement any interfaces nested within. Also, **private** interfaces cannot be implemented outside of their defining classes.

Initially, these features may seem like they are added strictly for syntactic consistency, but I generally find that once you know about a feature, you often discover places where it is useful.

Inner classes

It's possible to place a class definition within another class definition. This is called an *inner class*. The inner class is a valuable feature because it allows you to group classes that logically belong together and to control the visibility of one within the other. However, it's important to understand that inner classes are distinctly different from composition.

Often, while you're learning about them, the need for inner classes isn't immediately obvious. At the end of this section, after all of the syntax and semantics of inner classes have been described, you'll find examples that should make clear the benefits of inner classes.

You create an inner class just as you'd expect—by placing the class definition inside a surrounding class:

```
//: c08:Parcel1.java
// Creating inner classes.

public class Parcel1 {
```

```
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) {
      label = whereTo;
    }
    String readLabel() { return label; }
  }
  // Using inner classes looks just like
  // using any other class, within Parcel1:
  public void ship(String dest) {
    Contents c = new Contents();
    Destination d = new Destination(dest);
    System.out.println(d.readLabel());
  }
  public static void main(String[] args) {
    Parcel1 p = new Parcel1();
    p.ship("Tanzania");
  }
} ///:~
```

The inner classes, when used inside **ship( )**, look just like the use of any other classes. Here, the only practical difference is that the names are nested within **Parcel1**. You'll see in a while that this isn't the only difference.

More typically, an outer class will have a method that returns a reference to an inner class, like this:

```
//: c08:Parcel2.java
// Returning a reference to an inner class.

public class Parcel2 {
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
```

```
    Destination(String whereTo) {
      label = whereTo;
    }
    String readLabel() { return label; }
  }
  public Destination to(String s) {
    return new Destination(s);
  }
  public Contents cont() {
    return new Contents();
  }
  public void ship(String dest) {
    Contents c = cont();
    Destination d = to(dest);
    System.out.println(d.readLabel());
  }
  public static void main(String[] args) {
    Parcel2 p = new Parcel2();
    p.ship("Tanzania");
    Parcel2 q = new Parcel2();
    // Defining references to inner classes:
    Parcel2.Contents c = q.cont();
    Parcel2.Destination d = q.to("Borneo");
  }
} ///:~
```

If you want to make an object of the inner class anywhere except from within a non-**static** method of the outer class, you must specify the type of that object as *OuterClassName.InnerClassName*, as seen in **main( )**.

Inner classes and upcasting

So far, inner classes don't seem that dramatic. After all, if it's hiding you're after, Java already has a perfectly good hiding mechanism—just allow the class to be "friendly" (visible only within a package) rather than creating it as an inner class.

However, inner classes really come into their own when you start upcasting to a base class, and in particular to an **interface**. (The effect of producing an interface reference from an object that implements it is

essentially the same as upcasting to a base class.) That's because the inner class—the implementation of the **interface**—can then be completely unseen and unavailable to anyone, which is convenient for hiding the implementation. All you get back is a reference to the base class or the **interface**.

First, the common interfaces will be defined in their own files so they can be used in all the examples:

```
//: c08:Destination.java
public interface Destination {
  String readLabel();
} ///:~
```

```
//: c08:Contents.java
public interface Contents {
  int value();
} ///:~
```

Now **Contents** and **Destination** represent interfaces available to the client programmer. (The **interface**, remember, automatically makes all of its members **public**.)

When you get back a reference to the base class or the **interface**, it's possible that you can't even find out the exact type, as shown here:

```
//: c08:Parcel3.java
// Returning a reference to an inner class.

public class Parcel3 {
  private class PContents implements Contents {
    private int i = 11;
    public int value() { return i; }
  }
  protected class PDestination
      implements Destination {
    private String label;
    private PDestination(String whereTo) {
      label = whereTo;
    }
    public String readLabel() { return label; }
  }
```

```
  public Destination dest(String s) {
    return new PDestination(s);
  }
  public Contents cont() {
    return new PContents();
  }
}

class Test {
  public static void main(String[] args) {
    Parcel3 p = new Parcel3();
    Contents c = p.cont();
    Destination d = p.dest("Tanzania");
    // Illegal -- can't access private class:
    //! Parcel3.PContents pc = p.new PContents();
  }
} ///:~
```

Note that since **main( )** is in **Test**, when you want to run this program you don't execute **Parcel3**, but instead:

```
java Test
```

In the example, **main( )** must be in a separate class in order to demonstrate the privateness of the inner class **PContents**.

In **Parcel3**, something new has been added: the inner class **PContents** is **private** so no one but **Parcel3** can access it. **PDestination** is **protected**, so no one but **Parcel3**, classes in the **Parcel3** package (since **protected** also gives package access—that is, **protected** is also "friendly"), and the inheritors of **Parcel3** can access **PDestination**. This means that the client programmer has restricted knowledge and access to these members. In fact, you can't even downcast to a **private** inner class (or a **protected** inner class unless you're an inheritor), because you can't access the name, as you can see in **class Test**. Thus, the **private** inner class provides a way for the class designer to completely prevent any type-coding dependencies and to completely hide details about implementation. In addition, extension of an **interface** is useless from the client programmer's perspective since the client programmer cannot access any additional methods that aren't part of the **public interface**.

This also provides an opportunity for the Java compiler to generate more efficient code.

Normal (non-inner) classes cannot be made **private** or **protected**—only **public** or "friendly."

Inner classes in methods and scopes

What you've seen so far encompasses the typical use for inner classes. In general, the code that you'll write and read involving inner classes will be "plain" inner classes that are simple and easy to understand. However, the design for inner classes is quite complete and there are a number of other, more obscure, ways that you can use them if you choose: inner classes can be created within a method or even an arbitrary scope. There are two reasons for doing this:

1. As shown previously, you're implementing an interface of some kind so that you can create and return a reference.
2. You're solving a complicated problem and you want to create a class to aid in your solution, but you don't want it publicly available.

In the following examples, the previous code will be modified to use:

1. A class defined within a method
2. A class defined within a scope inside a method
3. An anonymous class implementing an interface
4. An anonymous class extending a class that has a nondefault constructor
5. An anonymous class that performs field initialization
6. An anonymous class that performs construction using instance initialization (anonymous inner classes cannot have constructors)

Although it's an ordinary class with an implementation, **Wrapping** is also being used as a common "interface" to its derived classes:

```
//: c08:Wrapping.java
public class Wrapping {
  private int i;
  public Wrapping(int x) { i = x; }
  public int value() { return i; }
} ///:~
```

You'll notice above that **Wrapping** has a constructor that requires an argument, to make things a bit more interesting.

The first example shows the creation of an entire class within the scope of a method (instead of the scope of another class):

```
//: c08:Parcel4.java
// Nesting a class within a method.

public class Parcel4 {
  public Destination dest(String s) {
    class PDestination
        implements Destination {
      private String label;
      private PDestination(String whereTo) {
        label = whereTo;
      }
      public String readLabel() { return label; }
    }
    return new PDestination(s);
  }
  public static void main(String[] args) {
    Parcel4 p = new Parcel4();
    Destination d = p.dest("Tanzania");
  }
} ///:~
```

The class **PDestination** is part of **dest( )** rather than being part of **Parcel4**. (Also notice that you could use the class identifier **PDestination** for an inner class inside each class in the same subdirectory without a name clash.) Therefore, **PDestination** cannot be accessed outside of **dest( )**. Notice the upcasting that occurs in the return statement—nothing comes out of **dest( )** except a reference to **Destination**, the base class. Of course, the fact that the name of the class

PDestination is placed inside **dest( )** doesn't mean that **PDestination** is not a valid object once **dest( )** returns.

The next example shows how you can nest an inner class within any arbitrary scope:

```
//: c08:Parcel5.java
// Nesting a class within a scope.

public class Parcel5 {
  private void internalTracking(boolean b) {
    if(b) {
      class TrackingSlip {
        private String id;
        TrackingSlip(String s) {
          id = s;
        }
        String getSlip() { return id; }
      }
      TrackingSlip ts = new TrackingSlip("slip");
      String s = ts.getSlip();
    }
    // Can't use it here! Out of scope:
    //! TrackingSlip ts = new TrackingSlip("x");
  }
  public void track() { internalTracking(true); }
  public static void main(String[] args) {
    Parcel5 p = new Parcel5();
    p.track();
  }
} ///:~
```

The class **TrackingSlip** is nested inside the scope of an **if** statement. This does not mean that the class is conditionally created—it gets compiled along with everything else. However, it's not available outside the scope in which it is defined. Other than that, it looks just like an ordinary class.

Anonymous inner classes

The next example looks a little strange:

```
//: c08:Parcel6.java
// A method that returns an anonymous inner class.

public class Parcel6 {
  public Contents cont() {
    return new Contents() {
      private int i = 11;
      public int value() { return i; }
    }; // Semicolon required in this case
  }
  public static void main(String[] args) {
    Parcel6 p = new Parcel6();
    Contents c = p.cont();
  }
} ///:~
```

The **cont( )** method combines the creation of the return value with the definition of the class that represents that return value! In addition, the class is anonymous—it has no name. To make matters a bit worse, it looks like you're starting out to create a **Contents** object:

```
return new Contents()
```

But then, before you get to the semicolon, you say, "But wait, I think I'll slip in a class definition":

```
return new Contents() {
  private int i = 11;
  public int value() { return i; }
};
```

What this strange syntax means is: "Create an object of an anonymous class that's inherited from **Contents**." The reference returned by the **new** expression is automatically upcast to a **Contents** reference. The anonymous inner-class syntax is a shorthand for:

```
class MyContents implements Contents {
  private int i = 11;
  public int value() { return i; }
}
return new MyContents();
```

In the anonymous inner class, **Contents** is created using a default constructor. The following code shows what to do if your base class needs a constructor with an argument:

```
//: c08:Parcel7.java
// An anonymous inner class that calls
// the base-class constructor.

public class Parcel7 {
  public Wrapping wrap(int x) {
    // Base constructor call:
    return new Wrapping(x) {
      public int value() {
        return super.value() * 47;
      }
    }; // Semicolon required
  }
  public static void main(String[] args) {
    Parcel7 p = new Parcel7();
    Wrapping w = p.wrap(10);
  }
} ///:~
```

That is, you simply pass the appropriate argument to the base-class constructor, seen here as the **x** passed in **new Wrapping(x)**. An anonymous class cannot have a constructor where you would normally call **super( )**.

In both of the previous examples, the semicolon doesn't mark the end of the class body (as it does in C++). Instead, it marks the end of the expression that happens to contain the anonymous class. Thus, it's identical to the use of the semicolon everywhere else.

What happens if you need to perform some kind of initialization for an object of an anonymous inner class? Since it's anonymous, there's no name to give the constructor—so you can't have a constructor. You can, however, perform initialization at the point of definition of your fields:

```
//: c08:Parcel8.java
// An anonymous inner class that performs
// initialization. A briefer version
```

```
// of Parcel5.java.

public class Parcel8 {
  // Argument must be final to use inside
  // anonymous inner class:
  public Destination dest(final String dest) {
    return new Destination() {
      private String label = dest;
      public String readLabel() { return label; }
    };
  }
  public static void main(String[] args) {
    Parcel8 p = new Parcel8();
    Destination d = p.dest("Tanzania");
  }
} ///:~
```

If you're defining an anonymous inner class and want to use an object that's defined outside the anonymous inner class, the compiler requires that the outside object be **final**. This is why the argument to **dest( )** is **final**. If you forget, you'll get a compile-time error message.

As long as you're simply assigning a field, the above approach is fine. But what if you need to perform some constructor-like activity? With *instance initialization*, you can, in effect, create a constructor for an anonymous inner class:

```
//: c08:Parcel9.java
// Using "instance initialization" to perform
// construction on an anonymous inner class.

public class Parcel9 {
  public Destination
  dest(final String dest, final float price) {
    return new Destination() {
      private int cost;
      // Instance initialization for each object:
      {
        cost = Math.round(price);
        if(cost > 100)
          System.out.println("Over budget!");
```

```
      }
      private String label = dest;
      public String readLabel() { return label; }
    };
  }
  public static void main(String[] args) {
    Parcel9 p = new Parcel9();
    Destination d = p.dest("Tanzania", 101.395F);
  }
} ///:~
```

Inside the instance initializer you can see code that couldn't be executed as part of a field initializer (that is, the **if** statement). So in effect, an instance initializer is the constructor for an anonymous inner class. Of course, it's limited; you can't overload instance initializers so you can have only one of these constructors.

The link to the outer class

So far, it appears that inner classes are just a name-hiding and code-organization scheme, which is helpful but not totally compelling. However, there's another twist. When you create an inner class, an object of that inner class has a link to the enclosing object that made it, and so it can access the members of that enclosing object—*without* any special qualifications. In addition, inner classes have access rights to all the elements in the enclosing class[3]. The following example demonstrates this:

```
//: c08:Sequence.java
// Holds a sequence of Objects.

interface Selector {
  boolean end();
  Object current();
  void next();
}
```

[3] This is very different from the design of *nested classes* in C++, which is simply a name-hiding mechanism. There is no link to an enclosing object and no implied permissions in C++.

```
public class Sequence {
  private Object[] obs;
  private int next = 0;
  public Sequence(int size) {
    obs = new Object[size];
  }
  public void add(Object x) {
    if(next < obs.length) {
      obs[next] = x;
      next++;
    }
  }
  private class SSelector implements Selector {
    int i = 0;
    public boolean end() {
      return i == obs.length;
    }
    public Object current() {
      return obs[i];
    }
    public void next() {
      if(i < obs.length) i++;
    }
  }
  public Selector getSelector() {
    return new SSelector();
  }
  public static void main(String[] args) {
    Sequence s = new Sequence(10);
    for(int i = 0; i < 10; i++)
      s.add(Integer.toString(i));
    Selector sl = s.getSelector();
    while(!sl.end()) {
      System.out.println(sl.current());
      sl.next();
    }
  }
} ///:~
```

The **Sequence** is simply a fixed-sized array of **object** with a class wrapped around it. You call **add( )** to add a new **object** to the end of the sequence (if there's room left). To fetch each of the objects in a **Sequence**, there's an interface called **Selector**, which allows you to see if you're at the **end( )**, to look at the **current( ) object**, and to move to the **next( ) object** in the **Sequence**. Because **Selector** is an **interface**, many other classes can implement the **interface** in their own ways, and many methods can take the **interface** as an argument, in order to create generic code.

Here, the **SSelector** is a **private** class that provides **Selector** functionality. In **main( )**, you can see the creation of a **Sequence**, followed by the addition of a number of **String** objects. Then, a **Selector** is produced with a call to **getSelector( )** and this is used to move through the **Sequence** and select each item.

At first, the creation of **SSelector** looks like just another inner class. But examine it closely. Note that each of the methods **end( )**, **current( ),** and **next( )** refer to **obs**, which is a reference that isn't part of **SSelector**, but is instead a **private** field in the enclosing class. However, the inner class can access methods and fields from the enclosing class as if they owned them. This turns out to be very convenient, as you can see in the above example.

So an inner class has automatic access to the members of the enclosing class. How can this happen? The inner class must keep a reference to the particular object of the enclosing class that was responsible for creating it. Then when you refer to a member of the enclosing class, that (hidden) reference is used to select that member. Fortunately, the compiler takes care of all these details for you, but you can also understand now that an object of an inner class can be created only in association with an object of the enclosing class. Construction of the inner class object requires the reference to the object of the enclosing class, and the compiler will complain if it cannot access that reference. Most of the time this occurs without any intervention on the part of the programmer.

static inner classes

If you don't need a connection between the inner class object and the outer class object, then you can make the inner class **static**. To understand the meaning of **static** when applied to inner classes, you must remember that the object of an ordinary inner class implicitly keeps a reference to the object of the enclosing class that created it. This is not true, however, when you say an inner class is **static**. A **static** inner class means:

1. You don't need an outer-class object in order to create an object of a **static** inner class.
2. You can't access an outer-class object from an object of a **static** inner class.

static inner classes are different than non-**static** inner classes in another way, as well. Fields and methods in non-**static** inner classes can only be at the outer level of a class, so non-**static** inner classes cannot have **static** data, **static** fields, or **static** inner classes. However, **static** inner classes can have all of these:

```
//: c08:Parcel10.java
// Static inner classes.

public class Parcel10 {
  private static class PContents
  implements Contents {
    private int i = 11;
    public int value() { return i; }
  }
  protected static class PDestination
      implements Destination {
    private String label;
    private PDestination(String whereTo) {
      label = whereTo;
    }
    public String readLabel() { return label; }
    // Static inner classes can contain
    // other static elements:
    public static void f() {}
```

```
    static int x = 10;
    static class AnotherLevel {
      public static void f() {}
      static int x = 10;
    }
  }
  public static Destination dest(String s) {
    return new PDestination(s);
  }
  public static Contents cont() {
    return new PContents();
  }
  public static void main(String[] args) {
    Contents c = cont();
    Destination d = dest("Tanzania");
  }
} ///:~
```

In **main( )**, no object of **Parcel10** is necessary; instead you use the normal syntax for selecting a **static** member to call the methods that return references to **Contents** and **Destination**.

As you will see shortly, in an ordinary (non-**static**) inner class, the link to the outer class object is achieved with a special **this** reference. A **static** inner class does not have this special **this** reference, which makes it analogous to a **static** method.

Normally you can't put any code inside an **interface**, but a **static** inner class can be part of an **interface**. Since the class is **static** it doesn't violate the rules for interfaces—the **static** inner class is only placed inside the namespace of the interface:

```
//: c08:IInterface.java
// Static inner classes inside interfaces.

public interface IInterface {
  static class Inner {
    int i, j, k;
    public Inner() {}
    void f() {}
  }
```

```
} ///:~
```

Earlier in this book I suggested putting a **main( )** in every class to act as a test bed for that class. One drawback to this is the amount of extra compiled code you must carry around. If this is a problem, you can use a **static** inner class to hold your test code:

```
//: c08:TestBed.java
// Putting test code in a static inner class.

public class TestBed {
  TestBed() {}
  void f() { System.out.println("f()"); }
  public static class Tester {
    public static void main(String[] args) {
      TestBed t = new TestBed();
      t.f();
    }
  }
} ///:~
```

This generates a separate class called **TestBed$Tester** (to run the program, you say **java TestBed$Tester**). You can use this class for testing, but you don't need to include it in your shipping product.

Referring to the outer class object

If you need to produce the reference to the outer class object, you name the outer class followed by a dot and **this**. For example, in the class **Sequence.SSelector**, any of its methods can produce the stored reference to the outer class **Sequence** by saying **Sequence.this**. The resulting reference is automatically the correct type. (This is known and checked at compile-time, so there is no run-time overhead.)

Sometimes you want to tell some other object to create an object of one of its inner classes. To do this you must provide a reference to the other outer class object in the **new** expression, like this:

```
//: c08:Parcel11.java
// Creating instances of inner classes.

public class Parcel11 {
```

```
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) {
      label = whereTo;
    }
    String readLabel() { return label; }
  }
  public static void main(String[] args) {
    Parcel11 p = new Parcel11();
    // Must use instance of outer class
    // to create an instances of the inner class:
    Parcel11.Contents c = p.new Contents();
    Parcel11.Destination d =
      p.new Destination("Tanzania");
  }
} ///:~
```

To create an object of the inner class directly, you don't follow the same form and refer to the outer class name **Parcel11** as you might expect, but instead you must use an *object* of the outer class to make an object of the inner class:

```
Parcel11.Contents c = p.new Contents();
```

Thus, it's not possible to create an object of the inner class unless you already have an object of the outer class. This is because the object of the inner class is quietly connected to the object of the outer class that it was made from. However, if you make a **static** inner class, then it doesn't need a reference to the outer class object.

Reaching outward from a multiply-nested class

[4]It doesn't matter how deeply an inner class may be nested—it can transparently access all of the members of all the classes it is nested within, as seen here:

```
//: c08:MultiNestingAccess.java
// Nested classes can access all members of all
// levels of the classes they are nested within.

class MNA {
  private void f() {}
  class A {
    private void g() {}
    public class B {
      void h() {
        g();
        f();
      }
    }
  }
}

public class MultiNestingAccess {
  public static void main(String[] args) {
    MNA mna = new MNA();
    MNA.A mnaa = mna.new A();
    MNA.A.B mnaab = mnaa.new B();
    mnaab.h();
  }
} ///:~
```

You can see that in **MNA.A.B**, the methods **g( )** and **f( )** are callable without any qualification (despite the fact that they are **private**). This example also demonstrates the syntax necessary to create objects of multiply-nested inner classes when you create the objects in a different

[4] Thanks again to Martin Danner.

class. The "**.new**" syntax produces the correct scope so you do not have to qualify the class name in the constructor call.

Inheriting from inner classes

Because the inner class constructor must attach to a reference of the enclosing class object, things are slightly complicated when you inherit from an inner class. The problem is that the "secret" reference to the enclosing class object *must* be initialized, and yet in the derived class there's no longer a default object to attach to. The answer is to use a syntax provided to make the association explicit:

```
//: c08:InheritInner.java
// Inheriting an inner class.

class WithInner {
  class Inner {}
}

public class InheritInner
    extends WithInner.Inner {
  //! InheritInner() {} // Won't compile
  InheritInner(WithInner wi) {
    wi.super();
  }
  public static void main(String[] args) {
    WithInner wi = new WithInner();
    InheritInner ii = new InheritInner(wi);
  }
} ///:~
```

You can see that **InheritInner** is extending only the inner class, not the outer one. But when it comes time to create a constructor, the default one is no good and you can't just pass a reference to an enclosing object. In addition, you must use the syntax

```
enclosingClassReference.super();
```

inside the constructor. This provides the necessary reference and the program will then compile.

Can inner classes be overridden?

What happens when you create an inner class, then inherit from the enclosing class and redefine the inner class? That is, is it possible to override an inner class? This seems like it would be a powerful concept, but "overriding" an inner class as if it were another method of the outer class doesn't really do anything:

```
//: c08:BigEgg.java
// An inner class cannot be overriden
// like a method.

class Egg {
  protected class Yolk {
    public Yolk() {
      System.out.println("Egg.Yolk()");
    }
  }
  private Yolk y;
  public Egg() {
    System.out.println("New Egg()");
    y = new Yolk();
  }
}

public class BigEgg extends Egg {
  public class Yolk {
    public Yolk() {
      System.out.println("BigEgg.Yolk()");
    }
  }
  public static void main(String[] args) {
    new BigEgg();
  }
} ///:~
```

The default constructor is synthesized automatically by the compiler, and this calls the base-class default constructor. You might think that since a **BigEgg** is being created, the "overridden" version of **Yolk** would be used, but this is not the case. The output is:

```
New Egg()
Egg.Yolk()
```

This example simply shows that there isn't any extra inner class magic going on when you inherit from the outer class. The two inner classes are completely separate entities, each in their own namespace. However, it's still possible to explicitly inherit from the inner class:

```
//: c08:BigEgg2.java
// Proper inheritance of an inner class.

class Egg2 {
  protected class Yolk {
    public Yolk() {
      System.out.println("Egg2.Yolk()");
    }
    public void f() {
      System.out.println("Egg2.Yolk.f()");
    }
  }
  private Yolk y = new Yolk();
  public Egg2() {
    System.out.println("New Egg2()");
  }
  public void insertYolk(Yolk yy) { y = yy; }
  public void g() { y.f(); }
}

public class BigEgg2 extends Egg2 {
  public class Yolk extends Egg2.Yolk {
    public Yolk() {
      System.out.println("BigEgg2.Yolk()");
    }
    public void f() {
      System.out.println("BigEgg2.Yolk.f()");
    }
  }
  public BigEgg2() { insertYolk(new Yolk()); }
  public static void main(String[] args) {
    Egg2 e2 = new BigEgg2();
    e2.g();
```

```
  }
} ///:~
```

Now **BigEgg2.Yolk** explicitly **extends Egg2.Yolk** and overrides its methods. The method **insertYolk( )** allows **BigEgg2** to upcast one of its own **Yolk** objects into the **y** reference in **Egg2**, so when **g( )** calls **y.f( )** the overridden version of **f( )** is used. The output is:

```
Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
```

The second call to **Egg2.Yolk( )** is the base-class constructor call of the **BigEgg2.Yolk** constructor. You can see that the overridden version of **f( )** is used when **g( )** is called.

Inner class identifiers

Since every class produces a **.class** file that holds all the information about how to create objects of this type (this information produces a "meta-class" called the **Class** object), you might guess that inner classes must also produce **.class** files to contain the information for *their* **Class** objects. The names of these files/classes have a strict formula: the name of the enclosing class, followed by a '**$**', followed by the name of the inner class. For example, the **.class** files created by **InheritInner.java** include:

```
InheritInner.class
WithInner$Inner.class
WithInner.class
```

If inner classes are anonymous, the compiler simply starts generating numbers as inner class identifiers. If inner classes are nested within inner classes, their names are simply appended after a '**$**' and the outer class identifier(s).

Although this scheme of generating internal names is simple and straightforward, it's also robust and handles most situations[5]. Since it is the standard naming scheme for Java, the generated files are automatically platform-independent. (Note that the Java compiler is changing your inner classes in all sorts of other ways in order to make them work.)

Why inner classes?

At this point you've seen a lot of syntax and semantics describing the way inner classes work, but this doesn't answer the question of why they exist. Why did Sun go to so much trouble to add this fundamental language feature?

Typically, the inner class inherits from a class or implements an **interface**, and the code in the inner class manipulates the outer class object that it was created within. So you could say that an inner class provides a kind of window into the outer class.

A question that cuts to the heart of inner classes is this: if I just need a reference to an **interface**, why don't I just make the outer class implement that **interface**? The answer is "If that's all you need, then that's how you should do it." So what is it that distinguishes an inner class implementing an **interface** from an outer class implementing the same **interface**? The answer is that you can't always have the convenience of **interface**s—sometimes you're working with implementations. So the most compelling reason for inner classes is:

> *Each inner class can independently inherit from an implementation. Thus, the inner class is not limited by whether the outer class is already inheriting from an implementation.*

Without the ability that inner classes provide to inherit—in effect—from more than one concrete or **abstract** class, some design and programming

---

[5] On the other hand, '$' is a meta-character to the Unix shell and so you'll sometimes have trouble when listing the **.class** files. This is a bit strange coming from Sun, a Unix-based company. My guess is that they weren't considering this issue, but instead thought you'd naturally focus on the source-code files.

problems would be intractable. So one way to look at the inner class is as the completion of the solution of the multiple-inheritance problem. Interfaces solve part of the problem, but inner classes effectively allow "multiple implementation inheritance." That is, inner classes effectively allow you to inherit from more than one non-**interface**.

To see this in more detail, consider a situation where you have two interfaces that must somehow be implemented within a class. Because of the flexibility of interfaces, you have two choices: a single class or an inner class:

```
//: c08:MultiInterfaces.java
// Two ways that a class can
// implement multiple interfaces.

interface A {}
interface B {}

class X implements A, B {}

class Y implements A {
  B makeB() {
    // Anonymous inner class:
    return new B() {};
  }
}

public class MultiInterfaces {
  static void takesA(A a) {}
  static void takesB(B b) {}
  public static void main(String[] args) {
    X x = new X();
    Y y = new Y();
    takesA(x);
    takesA(y);
    takesB(x);
    takesB(y.makeB());
  }
} ///:~
```

Of course, this assumes that the structure of your code makes logical sense either way. However, you'll ordinarily have some kind of guidance from the nature of the problem about whether to use a single class or an inner class. But without any other constraints, in the above example the approach you take doesn't really make much difference from an implementation standpoint. Both of them work.

However, if you have **abstract** or concrete classes instead of **interface**s, you are suddenly limited to using inner classes if your class must somehow implement both of the others:

```
//: c08:MultiImplementation.java
// With concrete or abstract classes, inner
// classes are the only way to produce the effect
// of "multiple implementation inheritance."

class C {}
abstract class D {}

class Z extends C {
  D makeD() { return new D() {}; }
}

public class MultiImplementation {
  static void takesC(C c) {}
  static void takesD(D d) {}
  public static void main(String[] args) {
    Z z = new Z();
    takesC(z);
    takesD(z.makeD());
  }
} ///:~
```

If you didn't need to solve the "multiple implementation inheritance" problem, you could conceivably code around everything else without the need for inner classes. But with inner classes you have these additional features:

1. The inner class can have multiple instances, each with its own state information that is independent of the information in the outer class object.

2. In a single outer class you can have several inner classes, each of which implement the same **interface** or inherit from the same class in a different way. An example of this will be shown shortly.

3. The point of creation of the inner class object is not tied to the creation of the outer class object.

4. There is no potentially confusing "is-a" relationship with the inner class; it's a separate entity.

As an example, if **Sequence.java** did not use inner classes, you'd have to say "a **Sequence** is a **Selector**," and you'd only be able to have one **Selector** in existence for a particular **Sequence**. Also, you can have a second method, **getRSelector( )**, that produces a **Selector** that moves backward through the sequence. This kind of flexibility is only available with inner classes.

Closures & Callbacks

A *closure* is a callable object that retains information from the scope in which it was created. From this definition, you can see that an inner class is an object-oriented closure, because it doesn't just contain each piece of information from the outer class object ("the scope in which it was created"), but it automatically holds a reference back to the whole outer class object, where it has permission to manipulate all the members, even **private** ones.

One of the most compelling arguments made to include some kind of pointer mechanism in Java was to allow *callbacks*. With a callback, some other object is given a piece of information that allows it to call back into the originating object at some later point. This is a very powerful concept, as you will see in Chapters 13 and 16. If a callback is implemented using a pointer, however, you must rely on the programmer to behave and not misuse the pointer. As you've seen by now, Java tends to be more careful than that, so pointers were not included in the language.

The closure provided by the inner class is a perfect solution; more flexible and far safer than a pointer. Here's a simple example:

```
//: c08:Callbacks.java
// Using inner classes for callbacks
```

```
interface Incrementable {
  void increment();
}

// Very simple to just implement the interface:
class Callee1 implements Incrementable {
  private int i = 0;
  public void increment() {
    i++;
    System.out.println(i);
  }
}

class MyIncrement {
  public void increment() {
    System.out.println("Other operation");
  }
  public static void f(MyIncrement mi) {
    mi.increment();
  }
}

// If your class must implement increment() in
// some other way, you must use an inner class:
class Callee2 extends MyIncrement {
  private int i = 0;
  private void incr() {
    i++;
    System.out.println(i);
  }
  private class Closure implements Incrementable {
    public void increment() { incr(); }
  }
  Incrementable getCallbackReference() {
    return new Closure();
  }
}

class Caller {
  private Incrementable callbackReference;
```

```
  Caller(Incrementable cbh) {
    callbackReference = cbh;
  }
  void go() {
    callbackReference.increment();
  }
}

public class Callbacks {
  public static void main(String[] args) {
    Callee1 c1 = new Callee1();
    Callee2 c2 = new Callee2();
    MyIncrement.f(c2);
    Caller caller1 = new Caller(c1);
    Caller caller2 =
      new Caller(c2.getCallbackReference());
    caller1.go();
    caller1.go();
    caller2.go();
    caller2.go();
  }
} ///:~
```

This example also provides a further distinction between implementing an interface in an outer class vs. doing so in an inner class. **Callee1** is clearly the simpler solution in terms of the code. **Callee2** inherits from **MyIncrement** which already has a different **increment( )** method which does something unrelated to that which is expected by the **Incrementable** interface. When **MyIncrement** is inherited into **Callee2**, **increment( )** can't be overridden for use by **Incrementable**, so you're forced to provide a separate implementation using an inner class. Also note that when you create an inner class you do not add to or modify the interface of the outer class.

Notice that everything except **getCallbackReference( )** in **Callee2** is **private**. To allow *any* connection to the outside world, the **interface Incrementable** is essential. Here you can see how **interface**s allow for a complete separation of interface from implementation.

The inner class **Closure** simply implements **Incrementable** to provide a hook back into **Callee2**—but a safe hook. Whoever gets the

Incrementable reference can, of course, only call **increment( )** and has no other abilities (unlike a pointer, which would allow you to run wild).

Caller takes an **Incrementable** reference in its constructor (although the capturing of the callback reference could happen at any time) and then, sometime latter, uses the reference to "call back" into the **Callee** class.

The value of the callback is in its flexibility—you can dynamically decide what functions will be called at run-time. The benefit of this will become more evident in Chapter 13, where callbacks are used everywhere to implement graphical user interface (GUI) functionality.

Inner classes & control frameworks

A more concrete example of the use of inner classes can be found in something that I will refer to here as a *control framework*.

An *application framework* is a class or a set of classes that's designed to solve a particular type of problem. To apply an application framework, you inherit from one or more classes and override some of the methods. The code you write in the overridden methods customizes the general solution provided by that application framework, in order to solve your specific problem. The control framework is a particular type of application framework dominated by the need to respond to events; a system that primarily responds to events is called an *event-driven system*. One of the most important problems in application programming is the graphical user interface (GUI), which is almost entirely event-driven. As you will see in Chapter 13, the Java Swing library is a control framework that elegantly solves the GUI problem and that heavily uses inner classes.

To see how inner classes allow the simple creation and use of control frameworks, consider a control framework whose job is to execute events whenever those events are "ready." Although "ready" could mean anything, in this case the default will be based on clock time. What follows is a control framework that contains no specific information about what it's controlling. First, here is the interface that describes any control event. It's an **abstract** class instead of an actual **interface** because the default

behavior is to perform the control based on time, so some of the implementation can be included here:

```
//: c08:controller:Event.java
// The common methods for any control event.
package c08.controller;

abstract public class Event {
  private long evtTime;
  public Event(long eventTime) {
    evtTime = eventTime;
  }
  public boolean ready() {
    return System.currentTimeMillis() >= evtTime;
  }
  abstract public void action();
  abstract public String description();
} ///:~
```

The constructor simply captures the time when you want the **Event** to run, while **ready( )** tells you when it's time to run it. Of course, **ready( )** could be overridden in a derived class to base the **Event** on something other than time.

action( ) is the method that's called when the **Event** is **ready( )**, and **description( )** gives textual information about the **Event**.

The following file contains the actual control framework that manages and fires events. The first class is really just a "helper" class whose job is to hold **Event** objects. You can replace it with any appropriate container, and in Chapter 9 you'll discover other containers that will do the trick without requiring you to write this extra code:

```
//: c08:controller:Controller.java
// Along with Event, the generic
// framework for all control systems:
package c08.controller;

// This is just a way to hold Event objects.
class EventSet {
  private Event[] events = new Event[100];
  private int index = 0;
```

```
  private int next = 0;
  public void add(Event e) {
    if(index >= events.length)
      return; // (In real life, throw exception)
    events[index++] = e;
  }
  public Event getNext() {
    boolean looped = false;
    int start = next;
    do {
      next = (next + 1) % events.length;
      // See if it has looped to the beginning:
      if(start == next) looped = true;
      // If it loops past start, the list
      // is empty:
      if((next == (start + 1) % events.length)
         && looped)
        return null;
    } while(events[next] == null);
    return events[next];
  }
  public void removeCurrent() {
    events[next] = null;
  }
}

public class Controller {
  private EventSet es = new EventSet();
  public void addEvent(Event c) { es.add(c); }
  public void run() {
    Event e;
    while((e = es.getNext()) != null) {
      if(e.ready()) {
        e.action();
        System.out.println(e.description());
        es.removeCurrent();
      }
    }
  }
} ///:~
```

EventSet arbitrarily holds 100 **Event**s. (If a "real" container from Chapter 9 is used here you don't need to worry about its maximum size, since it will resize itself). The **index** is used to keep track of the next available space, and **next** is used when you're looking for the next **Event** in the list, to see whether you've looped around. This is important during a call to **getNext( )**, because **Event** objects are removed from the list (using **removeCurrent( )**) once they're run, so **getNext( )** will encounter holes in the list as it moves through it.

Note that **removeCurrent( )** doesn't just set some flag indicating that the object is no longer in use. Instead, it sets the reference to **null**. This is important because if the garbage collector sees a reference that's still in use then it can't clean up the object. If you think your references might hang around (as they would here), then it's a good idea to set them to **null** to give the garbage collector permission to clean them up.

Controller is where the actual work goes on. It uses an **EventSet** to hold its **Event** objects, and **addEvent( )** allows you to add new events to this list. But the important method is **run( )**. This method loops through the **EventSet**, hunting for an **Event** object that's **ready( )** to run. For each one it finds **ready( )**, it calls the **action( )** method, prints out the **description( ),** and then removes the **Event** from the list.

Note that so far in this design you know nothing about exactly *what* an **Event** does. And this is the crux of the design; how it "separates the things that change from the things that stay the same." Or, to use my term, the "vector of change" is the different actions of the various kinds of **Event** objects, and you express different actions by creating different **Event** subclasses (in *Design Patterns* parlance, the **Event** subclasses represent the *Command Pattern*).

This is where inner classes come into play. They allow two things:

1. To create the entire implementation of a control-framework application in a single class, thereby encapsulating everything that's unique about that implementation. Inner classes are used to express the many different kinds of **action( )** necessary to solve the problem. In addition, the following example uses **private** inner classes so the implementation is completely hidden and can be changed with impunity.

2. Inner classes keep this implementation from becoming awkward, since you're able to easily access any of the members in the outer class. Without this ability the code might become unpleasant enough that you'd end up seeking an alternative.

Consider a particular implementation of the control framework designed to control greenhouse functions[6]. Each action is entirely different: turning lights, water, and thermostats on and off, ringing bells, and restarting the system. But the control framework is designed to easily isolate this different code. Inner classes allow you to have multiple derived versions of the same base class, **Event**, within a single class. For each type of action you inherit a new **Event** inner class, and write the control code inside of **action( )**.

As is typical with an application framework, the class **GreenhouseControls** is inherited from **Controller**:

```
//: c08:GreenhouseControls.java
// This produces a specific application of the
// control system, all in a single class. Inner
// classes allow you to encapsulate different
// functionality for each type of event.
import c08.controller.*;

public class GreenhouseControls
    extends Controller {
  private boolean light = false;
  private boolean water = false;
  private String thermostat = "Day";
  private class LightOn extends Event {
    public LightOn(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn on the light.
      light = true;
```

[6] For some reason this has always been a pleasing problem for me to solve; it came from my earlier book *C++ Inside & Out*, but Java allows a much more elegant solution.

```
    }
    public String description() {
      return "Light is on";
    }
  }
  private class LightOff extends Event {
    public LightOff(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here to
      // physically turn off the light.
      light = false;
    }
    public String description() {
      return "Light is off";
    }
  }
  private class WaterOn extends Event {
    public WaterOn(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      water = true;
    }
    public String description() {
      return "Greenhouse water is on";
    }
  }
  private class WaterOff extends Event {
    public WaterOff(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      water = false;
    }
    public String description() {
      return "Greenhouse water is off";
    }
```

```
  }
  private class ThermostatNight extends Event {
    public ThermostatNight(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      thermostat = "Night";
    }
    public String description() {
      return "Thermostat on night setting";
    }
  }
  private class ThermostatDay extends Event {
    public ThermostatDay(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Put hardware control code here
      thermostat = "Day";
    }
    public String description() {
      return "Thermostat on day setting";
    }
  }
  // An example of an action() that inserts a
  // new one of itself into the event list:
  private int rings;
  private class Bell extends Event {
    public Bell(long eventTime) {
      super(eventTime);
    }
    public void action() {
      // Ring every 2 seconds, 'rings' times:
      System.out.println("Bing!");
      if(--rings > 0)
        addEvent(new Bell(
          System.currentTimeMillis() + 2000));
    }
    public String description() {
      return "Ring bell";
```

```
    }
  }
  private class Restart extends Event {
    public Restart(long eventTime) {
      super(eventTime);
    }
    public void action() {
      long tm = System.currentTimeMillis();
      // Instead of hard-wiring, you could parse
      // configuration information from a text
      // file here:
      rings = 5;
      addEvent(new ThermostatNight(tm));
      addEvent(new LightOn(tm + 1000));
      addEvent(new LightOff(tm + 2000));
      addEvent(new WaterOn(tm + 3000));
      addEvent(new WaterOff(tm + 8000));
      addEvent(new Bell(tm + 9000));
      addEvent(new ThermostatDay(tm + 10000));
      // Can even add a Restart object!
      addEvent(new Restart(tm + 20000));
    }
    public String description() {
      return "Restarting system";
    }
  }
  public static void main(String[] args) {
    GreenhouseControls gc =
      new GreenhouseControls();
    long tm = System.currentTimeMillis();
    gc.addEvent(gc.new Restart(tm));
    gc.run();
  }
} ///:~
```

Note that **light**, **water**, **thermostat,** and **rings** all belong to the outer class **GreenhouseControls**, and yet the inner classes can access those fields without qualification or special permission. Also, most of the **action( )** methods involve some sort of hardware control, which would most likely involve calls to non-Java code.

Most of the **Event** classes look similar, but **Bell** and **Restart** are special. **Bell** rings, and if it hasn't yet rung enough times it adds a new **Bell** object to the event list, so it will ring again later. Notice how inner classes *almost* look like multiple inheritance: **Bell** has all the methods of **Event** and it also appears to have all the methods of the outer class **GreenhouseControls**.

Restart is responsible for initializing the system, so it adds all the appropriate events. Of course, a more flexible way to accomplish this is to avoid hard-coding the events and instead read them from a file. (An exercise in Chapter 11 asks you to modify this example to do just that.) Since **Restart( )** is just another **Event** object, you can also add a **Restart** object within **Restart.action( )** so that the system regularly restarts itself. And all you need to do in **main( )** is create a **GreenhouseControls** object and add a **Restart** object to get it going.

> This example should move you a long way toward appreciating the value of inner classes, especially when used within a control framework. However, in Chapter 13 you'll see how elegantly inner classes are used to describe the actions of a graphical user interface. By the time you finish that chapter you should be fully convinced.

Summary

Interfaces and inner classes are more sophisticated concepts than what you'll find in many OOP languages. For example, there's nothing like them in C++. Together, they solve the same problem that C++ attempts to solve with its multiple inheritance (MI) feature. However, MI in C++ turns out to be rather difficult to use, while Java interfaces and inner classes are, by comparison, much more accessible.

Although the features themselves are reasonably straightforward, the use of these features is a design issue, much the same as polymorphism. Over time, you'll become better at recognizing situations where you should use an interface, or an inner class, or both. But at this point in this book you

should at least be comfortable with the syntax and semantics. As you see these language features in use you'll eventually internalize them.

Exercises

Solutions to selected exercises can be found in the electronic document *The Thinking in Java Annotated Solution Guide*, available for a small fee from *www.BruceEckel.com*.

1. Prove that the fields in an **interface** are implicitly **static** and **final**.
2. Create an **interface** containing three methods, in its own **package**. Implement the interface in a different **package**.
3. Prove that all the methods in an **interface** are automatically **public**.
4. In **c07:Sandwich.java**, create an interface called **FastFood** (with appropriate methods) and change **Sandwich** so that it also implements **FastFood**.
5. Create three **interface**s, each with two methods. Inherit a new **interface** from the three, adding a new method. Create a class by implementing the new **interface** and also inheriting from a concrete class. Now write four methods, each of which takes one of the four **interface**s as an argument. In **main( )**, create an object of your class and pass it to each of the methods.
6. Modify Exercise 5 by creating an **abstract** class and inheriting that into the derived class.
7. Modify **Music5.java** by adding a **Playable interface**. Move the **play( )** declaration from **Instrument** to **Playable**. Add **Playable** to the derived classes by including it in the **implements** list. Change **tune( )** so that it takes a **Playable** instead of an **Instrument**.
8. Change Exercise 6 in Chapter 7 so that **Rodent** is an **interface**.
9. In **Adventure.java**, add an **interface** called **CanClimb**, following the form of the other interfaces.

10. Write a program that imports and uses **Month2.java**.

11. Following the example given in **Month2.java**, create an enumeration of days of the week.

12. Create an **interface** with at least one method, in its own package. Create a class in a separate package. Add a **protected** inner class that implements the **interface**. In a third package, inherit from your class and, inside a method, return an object of the **protected** inner class, upcasting to the **interface** during the return.

13. Create an **interface** with at least one method, and implement that **interface** by defining an inner class within a method, which returns a reference to your **interface**.

14. Repeat Exercise 13 but define the inner class within a scope within a method.

15. Repeat Exercise 13 using an anonymous inner class.

16. Create a **private** inner class that implements a **public interface**. Write a method that returns a reference to an instance of the **private** inner class, upcast to the **interface**. Show that the inner class is completely hidden by trying to downcast to it.

17. Create a class with a nondefault constructor and no default constructor. Create a second class that has a method which returns a reference to the first class. Create the object to return by making an anonymous inner class that inherits from the first class.

18. Create a class with a **private** field and a **private** method. Create an inner class with a method that modifies the outer class field and calls the outer class method. In a second outer class method, create an object of the inner class and call it's method, then show the effect on the outer class object.

19. Repeat Exercise 18 using an anonymous inner class.

20. Create a class containing a **static** inner class. In **main( )**, create an instance of the inner class.

21. Create an **interface** containing a **static** inner class. Implement this **interface** and create an instance of the inner class.

22. Create a class containing an inner class that itself contains an inner class. Repeat this using **static** inner classes. Note the names of the **.class** files produced by the compiler.

23. Create a class with an inner class. In a separate class, make an instance of the inner class.

24. Create a class with an inner class that has a nondefault constructor. Create a second class with an inner class that inherits from the first inner class.

25. Repair the problem in **WindError.java**.

26. Modify **Sequence.java** by adding a method **getRSelector( )** that produces a different implementation of the **Selector interface** that moves backward through the sequence from the end to the beginning.

27. Create an **interface U** with three methods. Create a class **A** with a method that produces a reference to a **U** by building an anonymous inner class. Create a second class **B** that contains an array of **U**. **B** should have one method that accepts and stores a reference to a **U** in the array, a second method that sets a reference in the array (specified by the method argument) to **null** and a third method that moves through the array and calls the methods in **U**. In **main( )**, create a group of **A** objects and a single **B**. Fill the **B** with **U** references produced by the **A** objects. Use the **B** to call back into all the **A** objects. Remove some of the **U** references from the **B**.

28. In **GreenhouseControls.java**, add **Event** inner classes that turn fans on and off.

29. Show that an inner class has access to the **private** elements of its outer class. Determine whether the reverse is true.

9: Collecting Your Objects

It's a fairly simple program that has only a fixed quantity of objects with known lifetimes.

In general, your programs will always be creating new objects based on some criteria that will be known only at the time the program is running. You won't know until run-time the quantity or even the exact type of the objects you need. To solve the general programming problem, you need to be able to create any number of objects, anytime, anywhere. So you can't rely on creating a named reference to hold each one of your objects:

```
MyObject myReference;
```

since you'll never know how many of these you'll actually need.

To solve this rather essential problem, C# has several ways to hold objects (or rather, references to objects). The built-in type is the array, which has been discussed before. Also, the C# System.Collections namespace has a reasonably complete set of *container classes* (also known as *collection classes*). Containers provide sophisticated ways to hold and manipulate your objects.

Containers open the door to the world of computing with data structures, where amazing results can be achieved by manipulating the abstract geometry of trees, vector spaces, and hyperplanes. While data structure programming lies outside of the workaday world of most programmers, it is very important in scientific, graphic, and game programming.

Arrays

Most of the necessary introduction to arrays was covered in Chapter #initialization and cleanup#, which showed how you define and initialize

an array. Holding objects is the focus of this chapter, and an array is just one way to hold objects. But there are a number of other ways to hold objects, so what makes an array special?

There are two issues that distinguish arrays from other types of containers: efficiency and type. The array is the most efficient way that C# provides to store and randomly access a sequence of objects (actually, object references). The array is a simple linear sequence, which makes element access fast, but you pay for this speed: when you create an array object, its size is fixed and cannot be changed for the lifetime of that array object. You might suggest creating an array of a particular size and then, if you run out of space, creating a new one and moving all the references from the old one to the new one. This is the behavior of the **ArrayList** class, which will be studied later in this chapter. However, because of the overhead of this size flexibility, an **ArrayList** is measurably less efficient than an array.

The **vector** container class in C++ *does* know the type of objects it holds, but it has a different drawback when compared with arrays in C#: the C++ **vector**'s **operator[]** doesn't do bounds checking, so you can run past the end[1]. In C#, you get bounds checking regardless of whether you're using an array or a container—you'll get an **IndexOutOfRangeException** if you exceed the bounds. As you'll learn in Chapter #Exceptions#, this type of exception indicates a programmer error, and thus you don't need to check for it in your code. As an aside, the reason the C++ **vector** doesn't check bounds with every access is speed—in C# you have the performance overhead of bounds checking all the time for both arrays and containers.

The other generic container classes that will be studied in this chapter, **ICollection, IList** and **IDictionary**, all deal with objects as if they had no specific type. That is, they treat them as type **object**, the root class of all classes in C#. This works fine from one standpoint: you need to build only one container, and any C# object will go into that container. This is the second place where an array is superior to the generic containers:

---

[1] It's possible, however, to ask how big the **vector** is, and the **at( )** method *does* perform bounds checking.

when you create an array, you create it to hold a specific type. This means that you get compile-time type checking to prevent you from putting the wrong type in, or mistaking the type that you're extracting. Of course, C# will prevent you from sending an inappropriate message to an object, either at compile-time or at run-time. So it's not much riskier one way or the other, it's just nicer if the compiler points it out to you, faster at run-time, and there's less likelihood that the end user will get surprised by an exception.

Typed generic classes (sometimes called "parameterized types" and sometimes just "generics") are not part of the initial .NET framework but will be. Unlike C++'s templates or Java's proposed extensions, Microsoft wishes to implement support for "parametric polymorphism" within the Common Language Runtime itself. Don Syme and Andrew Kennedy of Microsoft's Cambridge (England) Research Lab published papers in Spring 2001 on a proposed strategy and Anders Hjelsberg hinted at C#'s Spring 2002 launch that implementation was well under way.

For the moment, though, efficiency and type checking suggest using an array if you can. However, when you're trying to solve a more general problem arrays can be too restrictive. After looking at arrays, the rest of this chapter will be devoted to the container classes provided by C#.

Arrays are first-class objects

Regardless of what type of array you're working with, the array identifier is actually a reference to a true object that's created on the heap. This is the object that holds the references to the other objects, and it can be created either implicitly, as part of the array initialization syntax, or explicitly with a **new** expression. Part of the array object is the read-only **Length** property that tells you how many elements can be stored in that array object. For rectangular arrays, the **Length** property tells you the total size of the array, the **Rank** property tells you the number of dimensions in the array, and the **GetLength(int)** method will tell you how many elements are in the given rank.

The following example shows the various ways that an array can be initialized, and how the array references can be assigned to different array objects. It also shows that arrays of objects and arrays of primitives are

almost identical in their use. The only difference is that arrays of objects hold references, while arrays of primitives hold the primitive values directly.

```
//:c09:ArraySize.cs
// Initialization & re-assignment of arrays.

class Weeble {} // A small mythical creature

public class ArraySize {
  public static void Main() {
    // Arrays of objects:
    Weeble[] a; // Null reference
    Weeble[] b = new Weeble[5]; // Null references
    Weeble[,] c = new Weeble[2, 3]; //Rectangular
array
    Weeble[] d = new Weeble[4];
    for(int index = 0; index < d.Length; index++)
      d[index] = new Weeble();
    // Aggregate initialization:
    Weeble[] e = {
      new Weeble(), new Weeble(), new Weeble()
    };
    // Dynamic aggregate initialization:
    a = new Weeble[] {
      new Weeble(), new Weeble()
    };
    // Square dynamic aggregate initialization:
    c = new Weeble[,] {
        { new Weeble(), new Weeble(), new Weeble() },
        { new Weeble(), new Weeble(), new Weeble() }
    };

    System.Console.WriteLine("a.Length=" + a.Length);
    System.Console.WriteLine("b.Length = " +
b.Length);
    System.Console.WriteLine("c.Length = " +
c.Length);
    for(int rank = 0; rank < c.Rank; rank++){
        System.Console.WriteLine(
```

```
          "c.Length[{0}] = {1}", rank,
c.GetLength(rank));
    }
    // The references inside the array are
    // automatically initialized to null:
    for(int index = 0; index < b.Length; index++)
      System.Console.WriteLine("b[" + index + "]=" +
b[index]);
    System.Console.WriteLine("d.Length = " +
d.Length);
    System.Console.WriteLine("d.Length = " +
d.Length);
    a = d;
    System.Console.WriteLine("a.Length = " +
a.Length);

    // Arrays of primitives:
    int[] f; // Null reference
    int[] g = new int[5];
    int[] h = new int[4];
    for(int index = 0; index < h.Length; index++)
      h[index] = index*index;
    int[] i = { 11, 47, 93 };
    // Compile error: Use of unassigned local variable
'f'
    //!System.Console.WriteLine("f.Length=" +
f.Length);
    System.Console.WriteLine("g.Length = " +
g.Length);
    // The primitives inside the array are
    // automatically initialized to zero:
    for(int index = 0; index < g.Length; index++)
      System.Console.WriteLine("g[" + index + "]=" +
g[index]);
    System.Console.WriteLine("h.Length = " +
h.Length);
    System.Console.WriteLine("i.Length = " +
i.Length);
    f = i;
    System.Console.WriteLine("f.Length = " +
f.Length);
```

```
    f = new int[] { 1, 2 };
    System.Console.WriteLine("f.Length = " +
f.Length);
  }
} ///:~
```

Here's the output from the program:

```
a.Length=2
b.Length = 5
c.Length = 6
c.Length[0] = 2
c.Length[1] = 3
b[0]=
b[1]=
b[2]=
b[3]=
b[4]=
d.Length = 4
d.Length = 4
a.Length = 4
g.Length = 5
g[0]=0
g[1]=0
g[2]=0
g[3]=0
g[4]=0
h.Length = 4
i.Length = 3
f.Length = 3
f.Length = 2
```

The array **a** is initially just a **null** reference, and the compiler prevents you from doing anything with this reference until you've properly initialized it. The array **b** is initialized to point to an array of **Weeble** references, but no actual **Weeble** objects are ever placed in that array. However, you can still ask what the size of the array is, since **b** is pointing to a legitimate object. This brings up a slight drawback: you can't find out how many elements are actually *in* the array, since **Length** tells you only how many elements *can* be placed in the array; that is, the size of the array object, not the number of elements it actually holds. However, when an array object is created its references are automatically initialized to

null, so you can see whether a particular array slot has an object in it by checking to see whether it's **null**. Similarly, an array of primitives is automatically initialized to zero for numeric types, **(char)0** for **char**, and **false** for **bool**.

Array **c** shows the creation of the array object followed by the assignment of **Weeble** objects to all the slots in the array. Array **d** shows the "aggregate initialization" syntax that causes the array object to be created (implicitly with **new** on the heap, just like for array **c**) *and* initialized with **Weeble** objects, all in one statement.

The next array initialization could be thought of as a "dynamic aggregate initialization." The aggregate initialization used by **d** must be used at the point of **d**'s definition, but with the second syntax you can create and initialize an array object anywhere. For example, suppose **Hide()** is a method that takes an array of **Weeble** objects. You could call it by saying:

```
Hide(d);
```

but you can also dynamically create the array you want to pass as the argument:

```
Hide(new Weeble[] { new Weeble(), new Weeble() });
```

In some situations this new syntax provides a more convenient way to write code.

Rectangular arrays are initialized using nested arrays. Although a rectangular array is contiguous in memory, C#'s compiler will not allow you to ignore the dimensions; you cannot cast a flat array into a rectangular array or initialize a rectangular array in a "flat" manner.

The expression:

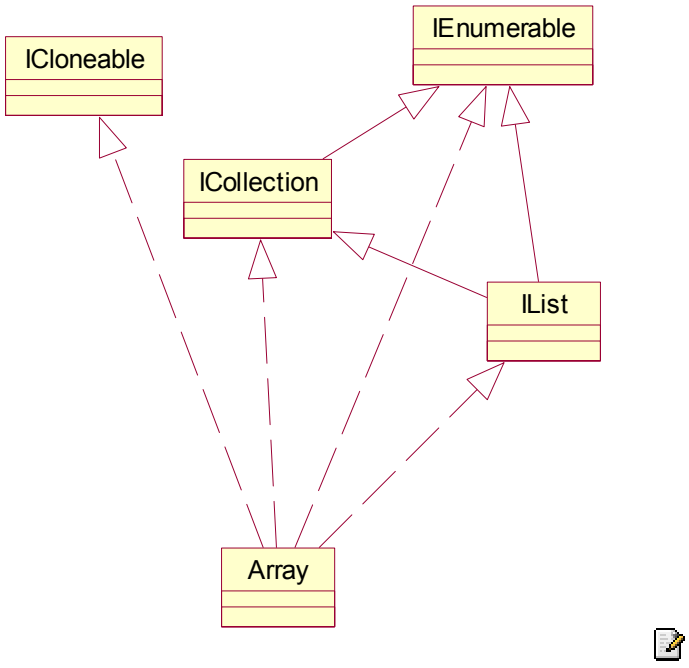
```
a = d;
```

shows how you can take a reference that's attached to one array object and assign it to another array object, just as you can do with any other type of object reference. Now both **a** and **d** are pointing to the same array object on the heap.

The second part of **ArraySize.cs** shows that primitive arrays work just like object arrays *except* that primitive arrays hold the primitive values directly.

The **Array** class

In **System.Collections**, you'll find the **Array** class, which has a variety of interesting properties and methods. **Array** is defined as implementing **ICloneable, IList, ICollection,** and **IEnumerable**. This is actually a pretty sloppy declaration, as **IList** is declared as extending **ICollection** and **IEnumerable**, while **ICollection** is itself declared as extending **IEnumerable** (Figure #)!



The **Array** class has some properties inherited from **ICollection** that are the same for all instances: **IsFixedSize** is always **true**, **IsReadOnly** and **IsSynchronized** are always **false.**

Array's Static Methods

The **Array** class has several useful static methods, which are illustrated in this program:

```
//:c09:ArrayStatics.cs
using System;
using System.Collections;

class Weeble {
    string name;
    internal string Name{
        get { return name;}
        set { name = value;}
    }
    internal Weeble(string name) {
        this.Name = name;
    }
}
class ArrayStatics {
    static string[] dayList = new string[]{
        "sunday", "monday", "tuesday", "wednesday",
"thursday", "friday", "saturday"
    };

    static string[,] famousCouples = new string[,]{
        { "George", "Martha"}, { "Napolean",
"Josephine"}, { "Westley","Buttercup"}
    };

    static Weeble[] weebleList = new Weeble[]{
        new Weeble("Pilot"), new Weeble("Firefighter")
    };

    public static void Main() {
        //Copying arrays
        Weeble[] newList = new
Weeble[weebleList.Length];
        Array.Copy(weebleList, newList,
weebleList.Length);
        newList[0] = new Weeble("Nurse");
```

```
        bool newReferences = newList[0] !=
weebleList[0];
        System.Console.WriteLine("New references ==  "
+ newReferences);
        //Copying a rectangular array works
        string[,] newSquareArray = new
string[famousCouples.GetLength(0),
famousCouples.GetLength(1)];
        Array.Copy(famousCouples, newSquareArray,
famousCouples.Length);

        //In-place sorting
        string[] sortedDays = new
string[dayList.Length];
        Array.Copy(dayList, sortedDays,
dayList.Length);
        Array.Sort(sortedDays);
        for (int i = 0; i < sortedDays.Length; i++) {
            System.Console.WriteLine("sortedDays[{0}]
= {1}", i, sortedDays[i]);
        }
        //Binary search of sorted 1-D Array
        int tuesdayIndex =
Array.BinarySearch(sortedDays, "tuesday");
        System.Console.WriteLine("dayList[{0}] ==
\"tuesday\"", tuesdayIndex);
        //! int georgeIndex =
Array.BinarySearch(famousCouples, "George"}; <-
Compile error

        //Reverse
        Array.Reverse(sortedDays);
        for (int i = 0; i < sortedDays.Length; i++) {
            System.Console.WriteLine("Reversed
sortedDays[{0}] = {1}", i, sortedDays[i]);
        }

        //Quickly erasing an array section, even if
multidimensional
        Array.Clear(famousCouples, 2, 3);
```

```
        for (int x = 0; x <
famousCouples.GetLength(0); x++)
            for (int y = 0; y <
famousCouples.GetLength(1); y++)

System.Console.WriteLine("FamousCouples[{0},{1}] =
{2}", x, y, famousCouples[x,y]);
    }
}///:~
```

After declaring a Weeble class (this time with a Name property to make them easier to distinguish), the **ArrayStatics** class declares several static arrays – **dayList** and **weebleList**, which are both one-dimensional, and the square **famousCouples** array.

Array.Copy() provides a fast way to copy an array (or a portion of it). The new array contains all new references, so changing a value in your new list will not change the value in your original, as would be the case if you did:

```
Weeble[] newList = weebleList;
newList[0] = new Weeble("Nurse");
```

Array.Copy() works with multidimensional arrays, too. The program uses the **GetLength(int)** method to allocate sufficient storage for the **newSquareArray**, but then uses the **famousCouples.Length** property to specify the size of the copy. Although **Copy()** seems to "flatten" multidimensional arrays, using arrays of different rank will throw a runtime **RankException**.

The static method **Array.Sort()** does an in-place sort of the array's contents and **BinarySearch()** provides an efficient search on a sorted array.

Array.Reverse() is self-explanatory, but **Array.Clear()** has the perhaps surprising behavior of slicing across multidimensional arrays. In the program, **Array.Clear(famousCouples, 2, 3)** treats the multidimensional **famousCouples** array as a flat array, setting to **null** the values of indices [1,0], [1,1], and [2,0].

Array element comparisons

How does **Array.Sort()** work? A problem with writing generic sorting code is that sorting must perform comparisons based on the actual type of the object. Of course, one approach is to write a different sorting method for every different type, but you should be able to recognize that this does not produce code that is easily reused for new types.

A primary goal of programming design is to "separate things that change from things that stay the same," and here, the code that stays the same is the general sort algorithm, but the thing that changes from one use to the next is the way objects are compared. So instead of hard-wiring the comparison code into many different sort routines, the technique of the *callback* is used. With a callback, the part of the code that varies from case to case is encapsulated inside its own class, and the part of the code that's always the same will call back to the code that changes. That way you can make different objects to express different ways of comparison and feed them to the same sorting code.

In C#, comparisons are done by calling back to the **CompareTo()** method of the **IComparable** interface. This method takes another **object** as an argument, and produces a negative value if the current object is less than the argument, zero if the argument is equal, and a positive value if the current object is greater than the argument.

Here's a class that implements **IComparable** and demonstrates the comparability by using **Array.Sort( )**:

```
//:c09:CompType.cs
// Implementing IComparable in a class.
using System;

public class CompType: IComparable {
  int i;
  int j;
  public CompType(int n1, int n2) {
    i = n1;
    j = n2;
  }
  public override string ToString() {
```

```
    return "[i = " + i + ", j = " + j + "]";
  }

  public int CompareTo(Object rv) {
    int rvi = ((CompType)rv).i;
    return (i < rvi ? -1 : (i == rvi ? 0 : 1));
  }

  private static Random r = new Random();

  private static void ArrayPrint(String s, Array a){
      System.Console.Write(s);
      foreach(Object o in a){
          System.Console.Write(o + ",");
      }
      System.Console.WriteLine();
  }

  public static void Main() {
    CompType[] a = new CompType[10];
    for(int i = 0; i < 10; i++){
        a[i] = new CompType(r.Next(100), r.Next(100));
    }
    ArrayPrint("Before sorting, a = ", a);
    Array.Sort(a);
    ArrayPrint("After sorting, a = ", a);
  }
} ///:~
```

When you define the comparison function, you are responsible for deciding what it means to compare one of your objects to another. Here, only the **i** values are used in the comparison, and the **j** values are ignored.

The **Main()** method creates a bunch of **CompType** objects that are initialized with random values and then sorted. If **Comparable** hadn't been implemented, then you'd get an InvalidOperationException thrown at runtime when you tried to call **Array.Sort( )**.

What? No bubbles?

In the not-so-distant past, the sort and search methods used in a program were a matter of constant debate and anguish. In the good old days, even the most trivial datasets had a good chance of being larger than RAM (or "core" as we used to say) and required intermediate reads and writes to storage devices that could take, yes, seconds to access (or, if the tapes needed to be swapped, minutes). So there was an enormous amount of energy put into worrying about internal (in-memory) versus external sorts, the stability of sorts, the importance of maintaining the input tape until the output tape was verified, the "operator dismount time," and so forth.

Nowadays, 99% of the time you can ignore the particulars of sorting and searching. In order to get a decent idea of sorting speed, this program requires an array of 1,000,000 elements, and still it executes in a matter of seconds:

```
//:c09:FastSort.cs
using System;

class Sortable : IComparable {
    int i;
    internal Sortable(int i) {
        this.i = i;
    }

    public int CompareTo(Object o) {
        try {
            Sortable s = (Sortable) o;
            return i = s.i;
        } catch (InvalidCastException) {
            throw new ArgumentException();
        }
    }
}

class SortingTester {
    static TimeSpan TimedSort(IComparable[] s){
        DateTime start = DateTime.Now;
```

```
        Array.Sort(s);
        TimeSpan duration = DateTime.Now - start;
        return duration;
    }
    public static void Main() {
        for (int times = 0; times < 10; times++) {
            Sortable[] s = new Sortable[1000000];
            for (int i = 0; i < s.Length; i++) {
                s[i] = new Sortable(i);
            }
            System.Console.WriteLine("Time to sort
already sorted array: " + TimedSort(s));
            Random rand = new Random();
            for (int i = 0; i < s.Length; i++) {
                s[i] = new Sortable(rand.Next());
            }
            System.Console.WriteLine("Time to sort
mixed up array: " + TimedSort(s));
        }
    }
}///:~
```

The results show that **Sort()** works faster on an already sorted array, which indicates that behind the scenes, it's probably using a merge sort instead of QuickSort. But the sorting algorithm is certainly less important than the fact that a computer that costs less than a thousand dollars can perform an in-memory sort of a million-item array! Moore's Law has made anachronistic an entire field of knowledge and debate that seemed, not that long ago, fundamental to computer programming.

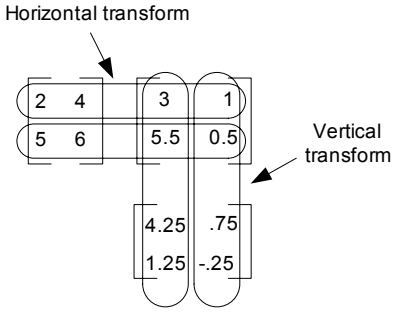
This is an important lesson for those who wish to have long careers in programming: never confuse the mastery of today's facts with preparation for tomorrow's changes. Within a decade, we will have multi-terabyte storage on the desktop, trivial access to distributed teraflop processing, and probably specialized access to quantum computers of significant capability. Eventually, although I doubt within a decade, there will be breakthroughs in user interfaces and we'll abandon the keyboard and the monitor for voice and gesture input and "augmented reality" glasses. Almost all the programming facts that hold today will be as useless as the knowledge of how to do an oscillating sort with criss-cross distribution. A programmer must never stand still.

Unsafe Arrays

Despite the preceding discussion of the steady march of technical obsolescence, the facts on the ground often agitate towards throwing away the benefits of safety and abstraction and getting closer to the hardware in order to boost performance. Often, the correct solution in this case will be to move out of C# altogether and into C++, a language which will continue for some time to be the best for the creation of device drivers and other close-to-the-metal components.

However, manipulating arrays can sometimes introduce bottlenecks in higher-level applications, such as multimedia applications. In such situations, unsafe code may be worthwhile. The basic impetus for using unsafe arrays is that you wish to manipulate the array as a contiguous block of memory, foregoing bounds checking.

Wavelet transforms are fascinating and their utility has hardly been scratched. The simplest transform is probably the two-dimensional Haar transform on a matrix of doubles. The Haar transform converts a list of values into the list's average and differences, so the list {2, 4} is transformed into {3, 1} == {(2 + 4) / 2, (2 + 4) / 2 – 2)}. A two-dimensional transform just transforms the rows and then the columns, so {{2, 4},{5,6}} becomes {{4.25, .75},{1.25, -0.25}}:



Wavelets have many interesting characteristics, including being the basis for some excellent compression routines, but are expensive to compute for arrays that are typical of multimedia applications, especially because to be useful they are usually computed $\log_2$(MIN(dimension size)) times per array!

The following program does such a transform in two different ways, one a safe method that uses typical C# code and the other using unsafe code.

```
//:c09:FastBitmapper1.cs
using System;
using System.IO;

namespace FastBitmapper{
    public interface Transform{
        void HorizontalTransform(double[,] matrix);
        void VerticalTransform(double[,] matrix);
    }

    public class Wavelet {
        public void Transform2D(double[,] matrix,
Transform t) {
            int minDimension = matrix.GetLength(0);
            if (matrix.GetLength(1) < minDimension)
                minDimension = matrix.GetLength(1);
            int levels = (int)
Math.Floor(Math.Log(minDimension, 2));
            Transform2D(matrix, levels, t);
        }

        public void Transform2D(double[,] matrix, int
steps, Transform tStrategy) {
            for (int i = 0; i < steps; i++) {
                tStrategy.HorizontalTransform(matrix);
                tStrategy.VerticalTransform(matrix);
            }
        }

        public void TestSpeed(Transform t) {
            Random rand = new Random();
            double[,] matrix = new double[2000,2000];
            for (int i = 0; i < matrix.GetLength(0);
i++)
                for (int j = 0; j <
matrix.GetLength(1); j++) {
                    matrix[i,j] = rand.NextDouble();
                }
```

```
            DateTime start = DateTime.Now;
            this.Transform2D(matrix, t);
            TimeSpan dur = DateTime.Now - start;
            System.Console.WriteLine("Transformation
with {0} took {1} ", t.GetType().Name, dur);

        }

        public static void Main() {
            Wavelet w = new Wavelet();
            for (int i = 0; i < 10; i++) {
                //Get things right first
                w.TestSpeed(new SafeTransform());
                w.TestSpeed(new UnsafeTransform());
            }
        }
    }


    internal class SafeTransform : Transform {
        private void Transform(double[] array) {
            int halfLength = array.Length >> 1;
            double[] avg = new double[halfLength];
            double[] diff = new double[halfLength];
            for (int pair = 0; pair < halfLength;
pair++) {
                double first = array[pair * 2];
                double next = array[pair * 2 + 1];
                avg[pair] = (first + next) / 2;
                diff[pair] = avg[pair] - first;
            }
            for (int pair = 0; pair < halfLength;
pair++) {
                array[pair] = avg[pair];
                array[pair + halfLength] = diff[pair];
            }
        }

        public void HorizontalTransform(double[,]
matrix) {
            int height = matrix.GetLength(0);
```

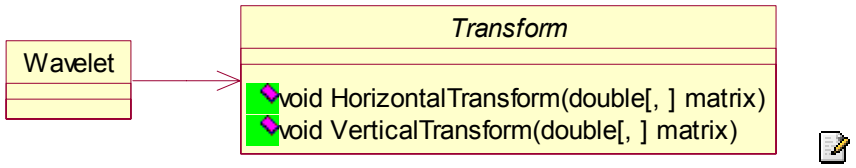
```
            int width = matrix.GetLength(1);
            double[] row = new double[width];
            for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                    row[j] = matrix[i, j];
                }
                Transform(row);
                for (int j = 0; j < width; j++) {
                    matrix[i,j] = row[j];
                }
            }
        }

        public void VerticalTransform(double[,]
matrix) {
            int height = matrix.GetLength(0);
            int length = matrix.GetLength(1);
            double[] colData = new double[height];
            for (int col = 0; col < length; col++) {
                for (int row = 0; row < height; row++)
{
                    colData[row] = matrix[row, col];
                }
                Transform(colData);
                for (int row = 0; row < height; row++)
{
                    matrix[row, col] = colData[row];
                }
            }
        }
    }
}///:~
```

Get things right…

The cardinal rule of performance programming is to first get the system operating properly and then worry about performance. The second rule is to always use a profiler to measure where your problems are, never go with a guess. In an object-oriented design, after discovering a hotspot, you should always break the problem out into an abstract data type (an interface) if it is not already. This will allow you to switch between

different implementations over time, confirming that your performance work is accomplishing something and that it is not diverging from your correct "safe" work.

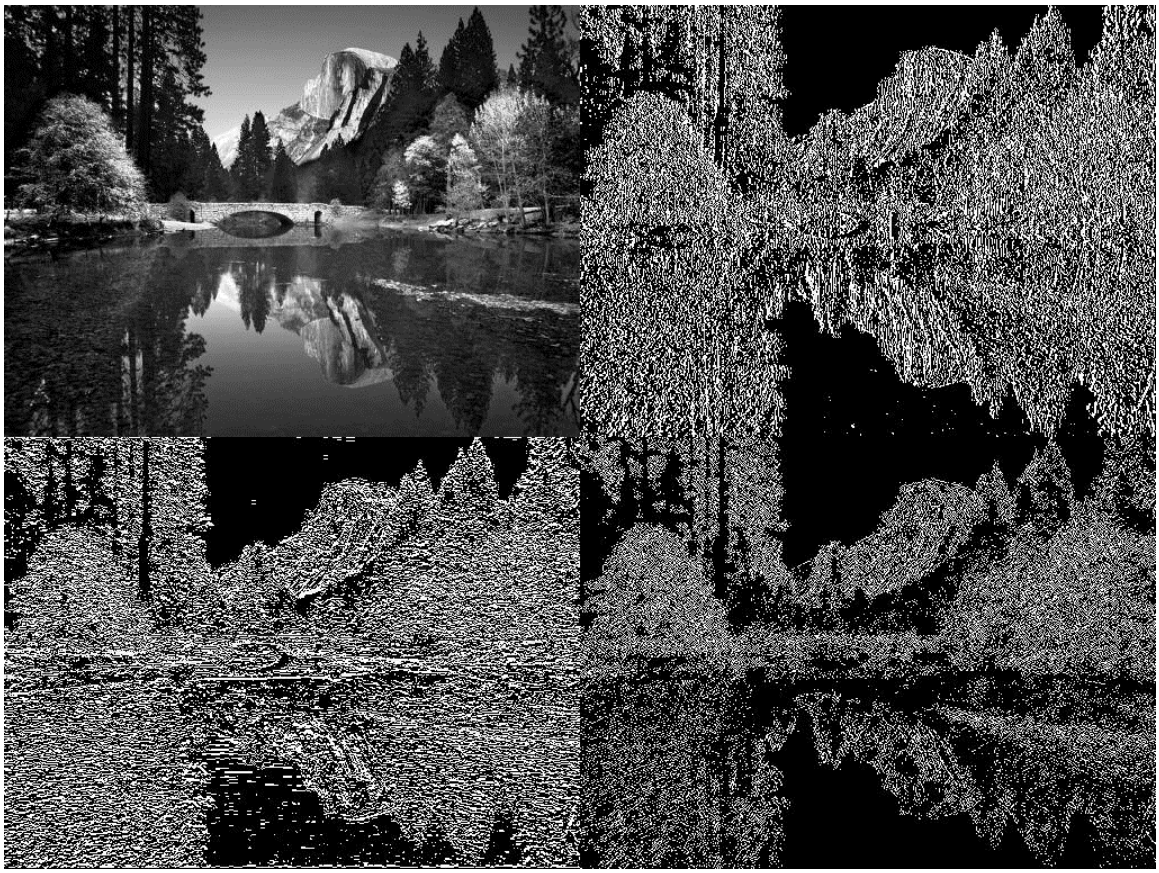
In this case, the Wavelet class uses an interface called Transform to perform the actual work:



The **Transform** interface contains two methods, each of which takes a rectangular array as a parameter and performs an in-place transformation; **HorizontalTransform()** converts a row of values into a row containing the averages and differences of the row, and **VerticalTransform()** performs a similar transformation on the columns of the array.

The **Wavelet** class contains two **Transform2D()** methods, the first of which takes a rectangular array and a **Transform**. The number of steps required to perform a full wavelet transform is calculated by first determining the minimum dimension of the passed-in matrix and then using the **Math.Log()** function to determine the base-2 magnitude of that dimension. **Math.Floor()** rounds that magnitude down and the result is cast to the integer number of steps that will be applied to the matrix. (Thus, an array with a minimum dimension of 4 would have 2 steps, an array with 1024 would have 9.)

The constructor then calls the second constructor, which takes the same parameters as the first plus the number of times to apply the wavelet (this is a separate constructor because during debugging a single wavelet step is much easier to comprehend than a fully processed one, as Figure # shows)



The **Transform2D()** method iterates **steps** times over the matrix, first performing a horizontal transform and then performing a vertical transform. Alternating between horizontal and vertical transforms is called the *nonstandard wavelet decomposition*. The *standard decomposition* performs **steps** horizontal transforms and then performs **steps** vertical transforms. With graphics anyway, the nonstandard decomposition allows for easier appreciation of the wavelet behavior; in Figure #, the upper-left quadrant is a half-resolution duplicate of the original, the upper-right a map of 1-pixel horizontal features, the lower-left a similar map of vertical features, and the lower-right a complete map of 1-pixel features. When the result is transformed again and again, the result has many interesting features, including being highly compressible with both lossless and lossy techniques.

The **TestSpeed()** method in **Wavelet** creates a 4,000,000-element square array, fills it with random doubles, and then calculates and prints the time necessary to perform a full wavelet transform on the result. The **Main()** method calls this **TestSpeed()** method 10 times in order to ensure that any transient operating system events don't skew the results. This first version of the code calls **TestSpeed()** with a **SafeTransform** – get things right and then get them fast.

The **SafeTransform** class has a private **Transform()** method which takes a one-dimensional array of doubles. It creates two arrays, **avg** and **diff** of half the width of the original. The first loop in **Transform()** moves across the source array, reading value pairs. It calculates and placed these pair's average and difference in the **avg** and **diff** arrays. After this loop finished, the values in **avg** are copied to the first half of the input array and the values in **diff** to the second half. After **Transform()** finishes, the input array now contains the values of a one-step, one-dimensional Haar transformation. (Note that the transform is fully reversible -- the original data can be restored by first adding and then subtracting a **diff** value to a corresponding **avg** value.)

SafeTransform.HorizontalTransform() determines the height of the passed-in matrix and, copies the values of each row into a one-dimensional array of doubles called **row**. Then the code calls the previously described **Transform()** method and copies the result back into the original two-dimensional matrix. When **HorizontalTransform()** is finished, the input matrix as a whole now contains a one-step, horizontal Haar transformation.

SafeTransform.VerticalTransform() uses a similar set of loops as **HorizontalTransform()**, but instead of copying rows from the input matrix, it copies the values in a column into a double array called **colData**, transforms that with **Transform()**, and copies the result back into the input matrix. When this finished, control returns to **Wavelet.Transform2D()**, and one step of the wavelet decomposition has been performed.

… Then Get Them Fast

Running this through a profiler (I use Intel's vTune) shows that a lot of time is spent in the **HorizontalTransform()** and **VerticalTransform()** methods in addition to the **Transform()** method itself. So, let's try to improve all three by using unsafe code:

```
//:c09:UnsafeTransform.cs
  internal class UnsafeTransform : Transform {
        unsafe private void Transform(double* array,
int length) {

//System.Console.WriteLine("UnsafeTransform({0}, {1}",
*array, length);
            double* pOriginalArray = array;
            int halfLength = length >> 1;
            double[] avg = new double[halfLength];
            double[] diff = new double[halfLength];
            for (int pair = 0; pair < halfLength;
pair++) {
                double first = *array;
                ++array;
                double next = *array;
                ++array;
                avg[pair] = (first + next) / 2;
                diff[pair] = avg[pair] - first;
            }
            for (int pair = 0; pair < halfLength;
pair++) {
                pOriginalArray[pair] = avg[pair];
                pOriginalArray[pair + halfLength] =
diff[pair];
            }
        }

        unsafe public void
HorizontalTransform(double[,] matrix) {
            int height = matrix.GetLength(0);
            int width = matrix.GetLength(1);
            fixed(double* pMatrix = matrix) {
                double* pOffset = pMatrix;
```

```
                for (int row = 0; row < height; row++)
{
                    Transform(pOffset, width);
                    pOffset += width;
                }
            }
        }

        unsafe public void VerticalTransform(double[,]
matrix) {
            fixed(double* pMatrix = matrix) {
                int height = matrix.GetLength(0);
                int length = matrix.GetLength(1);
                double[] colData = new double[height];
                for (int col = 0; col < length; col++)
{
                    for (int row = 0; row < height;
row++) {
                        colData[row] = pMatrix[col +
length * row];
                    }
                    fixed(double* pColData = colData)
{
                        Transform(pColData, height);
                    }
                    for (int row = 0; row < height;
row++) {
                        pMatrix[col + length * row] =
colData[row];
                    }
                }
            }
        }///:~
```

First, notice that **UnsafeTransform** has the same structure as **SafeTransform**, a private **Transform()** function in addition to the public methods which implement **Transform**. This is by no means necessary, but it's a good starting place for optimization.

UnsafeTransform.Transform() has a signature unlike any C# signature discussed before: **unsafe private void Transform(double***

array, int length); When a method is declared unsafe, C# allows a new type of variable, called a pointer. A pointer contains a memory address at which a value of the specified type is located. So the variable **array** contains not a double value such as 0.2 or 234.28, but a memory location someplace in the runtime, the contents of which are interpreted as a double. Adding 1 to **array** does not change it to 1.2 or 235.28 but rather changes the memory location to point to the next location in memory that's big enough to hold a **double**. Such "pointer arithmetic" is marginally more efficient than using a C# array, but even small differences add up when applied to a 4,000,000 item array!

The first line in **UnsafeTransform.Transform()** initializes another pointer variable **pOriginalArray** with the original value in **array**, whose value is going to change. The declaration of the **avg** and **diff** arrays and the first loop are identical with what was done in **SafeTransform.Transform()**, except that this time we use the value of the passed-in **length** variable to calculate the value of **halfLength** (in **SafeTransform.Transform()**, we used the **Length** property of the passed-in array, but pointers don't have such a property, so we need the extra parameter. The next lines, though, are quite different:

```
double first = *array;
++array;
double next = *array;
++array;
```

When applied to a pointer variable, the * operator retrieves the value that is stored at that address (the mnemonic is "star = stored"). So the **first** double is assigned the value of the double at **array**'s address value. Then, we use pointer arithmetic on **array** so that it skips over a double's worth of memory, read the value there as a double and assign it to **next** and increment **array** again. The values of **avg** and **diff** are calculated just as they were in **SafeTransform.Transform()**.

So the big difference in this loop is that instead of indexing in to an array of **double**s of a certain length, we've incremented a pointer to **double**s **length** times, and interpreted the memory of where we were pointing at as a series of **double**s. There's been no bounds or type checking on the value of our **array** pointer, so if this method were called with either

array set incorrectly or with a wrong **length**, this loop would blithely read whatever it happened to be pointing at.

Such a situation might be hard to track down, but the final loop in **Unsafe.Transform()** would probably not go undetected. A feature of pointers is that you can use array notation to indicate an offset in memory. Thus, in this loop, we write back into the region of memory at **pOriginalArray** large enough to contain **length** doubles. Writing into an invalid region of memory is a pretty sure way to cause a crash. So it behooves us to make sure that **Unsafe.Transform()** is only called properly.

Unsafe.HorizontalTransform() takes a two-dimensional rectangular array of doubles called **matrix**. Before calling **Unsafe.Transform()**, which takes a pointer to a double, the **matrix** must be "pinned" in memory. The .NET garbage collector is normally free to move objects about, because the garbage collector has the necessary data to determine every reference to that object (indeed, tracking those references is the very essence of garbage collection!). But when a pointer is involved, it's not safe to move references; in our case, the loops in **Transform** both read and write a large block of memory based on the original passed-in address.

The line **fixed(double* pMatrix = matrix)** pins the rectangular array **matrix** in memory and initializes a pointer to the beginning of that memory. Pointers initialized in a **fixed** declaration are read-only and for the purposes of pointer arithmetic, we need the next line to declare another pointer variable **pOffset** and initialize it to the value of **pMatrix**.

Notice that unlike **SafeTransform.HorizontalTransform()**, we do not have a temporary one-dimensional **row** array which we load before calling **Transform()** and copy from after. Instead, the main loop in **HorizontalTransform()** calls **Transform()** with its pointer of **pOffset** and its length set to the previously calculated width of the input **matrix**. Then, we use pointer arithmetic to jump **width** worth of **double**s in memory. In this way, we are exploiting the fact that we know that a rectangular array is, behind-the-scenes, a contiguous chunk of

memory. The line **pOffset += width;** is significantly faster than the 8 lines of safe code it replaces.

In **UnsafeTransform.VerticalTransform()**, though, no similar shortcut comes to mind and the code is virtually identical to that in **SafeTransform.VerticalTransform()** except that we still need to pin **matrix** in order to get the **pMatrix** pointer to pass to **Transform()**.

If we go back to Wavelet.Main() and uncomment the line that calls **TestSpeed()** with a **new UnsafeTransform()**, we're almost ready to go. However, the C# compiler requires a special flag in order to compile source that contains unsafe code. On the command-line, this flag is /unsafe , while in Visual Studio .NET, the option is found by right-clicking on the Project in the Solution Explorer and choosing Properties / Configuration Properties / Build and setting "Allow unsafe code blocks" to **true**.

On my machines, **UnsafeTransform** runs about 50% faster than **SafeTransform** in debugging mode, and is about 20% superior when optimizations are turned on. Hardly the stuff of legends, but in a core algorithm, perhaps worth the effort.

There's only one problem. This managed code implementation runs 40% faster than **UnsafeTransform**! Can you reason why?:

```
//:c09:InPlace.cs
    internal class InPlace : Transform {
        int length;
        int height;
        int halfLength;
        int halfHeight;
        //Half the length of longer dimension
        double[] diff = null;

        private void LazyInit(double[,] matrix) {
            height = matrix.GetLength(0);
            length = matrix.GetLength(1);
            halfLength = length >> 1;
            halfHeight = height >> 1;
            if (halfHeight < halfLength) {
                diff = new double[halfLength];
```

```
            } else {
                diff = new double[halfHeight];
            }
        }

        public void HorizontalTransform(double[, ]
matrix) {
            if (diff == null) {
                LazyInit(matrix);
            }
            for (int i = 0; i < height; i++) {
                HTransform(matrix, i);
            }

        }

        public void VerticalTransform(double[, ]
matrix) {
            if (diff == null) {
                LazyInit(matrix);
            }

            for (int col = 0; col < length; col++) {
                VTransform(matrix, col);
            }
        }

        private void HTransform(double[,] matrix, int
row) {
            for (int pair = 0; pair < halfLength;
pair++) {
                double first = matrix[row, pair * 2];
                double next = matrix[row, pair * 2 +
1];
                double avg = (first + next) / 2;
                matrix[row, pair * 2] = avg;
                diff[pair] = avg - first;
            }
            for (int pair = 0; pair < halfLength;
pair++) {
```

```
                matrix[row, pair + halfLength] =
diff[pair];
            }
        }

        private void VTransform(double[,] matrix, int
col) {
            for (int pair = 0; pair < halfHeight;
pair++) {
                double first = matrix[pair * 2, col];
                double next = matrix[pair * 2 + 1,
col];
                double avg = (first + next) / 2;
                matrix[pair * 2, col] = avg;
                diff[pair] = avg - first;
            }
            for (int pair = 0; pair < halfHeight;
pair++) {
                matrix[pair + halfHeight, col] =
diff[pair];
            }
        }
    }///:~
```

InPlace removes loops and allocations of temporary objects (like the **avg** and **diff** arrays) at the cost of clarity. In **SafeTransform**, the Haar algorithm of repeated averaging and differencing is pretty easy to follow just from the code; I daresay that a first-time reader of **InPlace** might not intuit, for instance, that the contents of the **diff** array are strictly for temporary storage.

Notice that both **HorizontalTransform()** and **VerticalTransform()** check to see if **diff** is null and call **LazyInit()** if it is not. Some might say "Well, we know that **HorizontalTransform()** is called first, so the check in **VerticalTransform()** is superfluous." But if we were to remove the check from **VerticalTransform()**, we would be changing the design contract of the **Transform()** interface to include "You must call **HorizontalTransform()** before calling **VerticalTransform()**."

Changing a design contract is not the end of the world, but it should always be given some thought. When a contract requires that method **A()**

be called before method **B()**, the two methods are said to be "sequence coupled." Sequence coupling is usually acceptable (unlike, say, "internal data coupling" where one class directly writes to another class's variables without using properties or methods to access the variables). Given that the check in **VerticalTransform**() is not within a loop, changing the contract doesn't seem worth what will certainly be an unmeasurably small difference in performance.

Array summary

To summarize what you've seen so far, the first and easiest choice to hold a group of objects of a known size is an array. Arrays are also the natural data structure to use if the way you wish to access the data is by a simple index, or if the data is naturally "rectangular" in its form. In the remainder of this chapter we'll look at the more general case, when you don't know at the time you're writing the program how many objects you're going to need, or if you need a more sophisticated way to store your objects. C# provides a library of *collection classes* to solve this problem, the basic types of which are **IList** and **IDictionary**. You can solve a surprising number of problems using these tools!

Among their other characteristics, the C# collection classes will automatically resize themselves. So, unlike arrays, you can put in any number of objects and you don't need to worry about how big to make the container while you're writing the program.

Introduction to data structures

Container classes are one of the most powerful tools for raw development because they provide an entry into the world of data structure programming. An interesting fact of programming is that the hardest challenges often boil down to selecting a data structure and applying a handful of simple operations to it. Object orientation makes it trivial to create data structures that work with abstract data types (i.e., a collection class is written to work with type **object** and thereby works with everything).

The .NET System.Collections namespace takes the issue of "holding your objects" and divides it into two distinct concepts:

1. **IList**: a group of individual elements, often with some rule applied to them. An **IList** must hold the elements in a particular sequence. , and a **Set** cannot have any duplicate elements. (Note that the .NET Framework does not supply either a *set*, which is a Collection without duplicates, nor a *bag*, which is an undordered Collection.)

2. **IDictionary**: a group of key-value object pairs (also called *Maps*). Strictly speaking, an **IDictionary** contains **DictionaryEntry** structures, which themselves contain the two references (in the **Key** and **Value** properties). The **Key** property cannot be null and must be unique, while the **Value** entry may be null or may point to a previously referenced object. You can access any of these parts of the **IDictionary** structure – you can get the **DictionaryEntry** values, the set of **Key**s or the collection of **Value**s. Dictionaries, like arrays, can easily be expanded to multiple dimensions without adding new concepts: you simply make an **IDictionary** whose values are of type **IDictionary** (and the values of *those* dictionaries can be dictionaries, etc.)

Queues and Stacks

For scheduling problems and other programs that need to deal with elements in order, but which when done discard or hand-off the elements to other components, you'll want to conside a Queue or a Stack.

A queue is a data structure which works like a line in a bank; the first to arrive is the first to be served.

A stack is often compared to a cafeteria plate-dispenser – the last object to be added is the first to be accessed. This example uses this metaphor to show the basic functions of a queue and a stack:

```
//:c09:QueueAndStack.cs
//Demonstrate time-of-arrival data structures
using System;
using System.Collections;
```

```
class Customer{
    string name;
    public string Name{
        get{ return name; }
    }

    PlateDispenser p;

    internal Customer(String name, PlateDispenser p){
        this.name = name;
        this.p = p;
    }

    internal void GetPlate(){
        string plate = p.GetPlate();
        System.Console.WriteLine(
            name + " got " + plate);
    }
}
class PlateDispenser{
    Stack dispenser = new Stack();
    internal void Fill(int iToPush){
        for(int i = 0; i < iToPush; i++){
            string p = "Plate #" + i;
            System.Console.WriteLine("Loading " + p);
            dispenser.Push(p);
        }
    }

    internal string GetPlate(){
        return (string) dispenser.Pop();
    }
}

class Teller{
    Queue line = new Queue();
    internal void EnterLine(Customer c){
       line.Enqueue(c);
    }
    internal void Checkout(){
        Customer c = (Customer) line.Dequeue();
```

```
        System.Console.WriteLine(
            "Checking out: " + c.Name);
    }
}
class Cafeteria{
    PlateDispenser pd = new PlateDispenser();
    Teller t = new Teller();

    public static void Main(){
        new Cafeteria();
    }

    public Cafeteria(){
        pd.Fill(4);
        Customer[] c = new Customer[4];
        for(int i = 0; i < 4; i++){
            c[i] = new Customer("Customer #" + i, pd);
            c[i].GetPlate();
        }
        for(int i = 0; i < 4; i++){
            t.EnterLine(c[i]);
        }
        for(int i = 0; i < 4; i++){
            t.Checkout();
        }
    }
}///:~
```

First, the code specifies that it will be using types from the System and System.Collection namespaces. Then, the **Customer** class has a name and a reference to a **PlateDispenser** object. These references are passed in the **Customer** constructor. Finally, **Customer.GetPlate()** retrieves a plate from the **PlateDispenser** and prints out the name of the customer and the identifier of the plate.

The **PlateDispenser** object contains an internal reference to a **Stack**. When **PlateDispenser.Fill()** is called, a unique string is created and **Stack.Push()** places it on the top (or front) of the stack. Similarly, **PlateDispenser.GetPlate()** uses **Stack.Pop()** to get the object at the stack's top.

The **Teller** class has a reference to a **Queue** object. **Teller.EnterLine()** calls **Queue.Enqueue()** and **Teller.Checkout()** calls **Queue.Dequeue()**. Since the only objects placed in the queue are of type **Customer**, it's safe for the reference returned by **Queue.Dequeue()** to be cast to a **Customer**, and the name printed to the Console.

Finally, the **Cafeteria** class brings it all together. It contains a **PlateDispenser** and a **Teller**. The constructor fills the plate dispenser and creates some customers, who get plates, get in line for the teller, and check out. The output looks like this:

```
Loading Plate #0
Loading Plate #1
Loading Plate #2
Loading Plate #3
Customer #0 got Plate #3
Customer #1 got Plate #2
Customer #2 got Plate #1
Customer #3 got Plate #0
Checking out: Customer #0
Checking out: Customer #1
Checking out: Customer #2
Checking out: Customer #3
```

As you can see, the order in which the plates are dispensed is the reverse of the order in which they were placed in the **PlateDispenser**'s **Stack**.

What happens if you call **Pop()** or **Dequeue()** on an empty collection? In both situations you'll get an **InvalidOperationException** with an explicit message that the stack or queue is empty.

Stacks and queues are just the thing for scheduling problems, but if you need to choose access on more than a time-of-arrival basis, you'll need another data structure.

ArrayList

If a numeric index is all you need, the first thing that you'll consider is an Array, of course. But if you don't know the exact number of objects that you'll need to store, consider ArrayList.

Like the other collection classes, ArrayList has some very handy static methods you can use when you want to ensure certain characteristics of the underlying collection. The static methods **ArrayList.FixedSize()** and **ArrayList.ReadOnly()** return their **ArrayList** arguments wrapped in specialized handles that enforce these restrictions. However, care must be taken to discard any references to the original argument to these methods, because the inner ArrayList can get around the restrictions, as this example shows:

```
//:c09:ArrayListStatics.cs
using System;
using System.Collections;

public class ArrayListStatics{
    public static void Main(){
        ArrayList al = new ArrayList();
        Random rand = new Random();
        int iToAdd = 50 + rand.Next(50);
        for(int i = 0; i < iToAdd; i++){
            string s = "String #" + i;
            al.Add(s);
        }
        ArrayList noMore = ArrayList.FixedSize(al);
        try{
            noMore.Add("This won't work");
        }catch(Exception ex){
            System.Console.WriteLine(ex);
        }
        ArrayList untouchable =
           ArrayList.ReadOnly(al);
        try{
            untouchable[0] = "This won't work";
        }catch(Exception ex){
            System.Console.WriteLine(ex);
        }
        //But restrictions do not apply to original
        al[0] = "Modified";
        System.Console.WriteLine(
           "Untouchable[0] = " + untouchable[0]);
        int originalCount = noMore.Count;
        System.Console.WriteLine(
```

```
            "Size of noMore {0} != {1}",
            originalCount, noMore.Count);
    }
}///:~
```

While the operations on the wrapped arrays will raise **NotSupportedExceptions** (which are caught and printed to the console), a change to the original **al** ArrayList is reflected in **Untouchable** and the size of **noMore** can be increased! Another interesting static method of ArrayList is **Synchronized**, which will be discussed in Chapter #threading#.

BitArray

A **BitArray** is used if you want to efficiently store a lot of on-off or true-false information. It's efficient only from the standpoint of size; if you're looking for efficient access, it is slightly slower than using an array of some native type.

A normal container expands as you add more elements, but with **BitArray**, you must set the **Length** property to be sufficient to hold as many as you need. The constructor to **BitArray** takes an integer which specifies the initial capacity (there are also constructors which copy from an existing **BitArray**, from an array of bools, or from the bit-values of an array of bytes or ints).

The following example shows how the **BitArray** works:

```
//:c09:Bits.cs
// Demonstration of BitSet.
using System;
using System.Collections;

public class Bits {
   static void PrintBitArray(BitArray b) {
      System.Console.WriteLine("bits: " + b);
      string bbits = "";
      for (int j = 0; j < b.Length ; j++)
         bbits += (b[j] ? "1" : "0");
      System.Console.WriteLine(
          "bit pattern: " + bbits);
```

```
}
public static void Main() {
   Random rand = new Random();
   // Take the LSB of Next():
   byte bt = (byte)rand.Next();
   BitArray bb = new BitArray(8);
   for (int i = 7; i >=0; i--)
      if (((1 << i) &  bt) != 0)
         bb.Set(i, true);
      else
         bb.Set(i, false);
   System.Console.WriteLine("byte value: " + bt);
   PrintBitArray(bb);

   short st = (short)rand.Next();
   BitArray bs = new BitArray(16);
   for (int i = 15; i >=0; i--)
      if (((1 << i) &  st) != 0)
         bs.Set(i, true);
      else
         bs.Set(i, false);
   System.Console.WriteLine("short value: " + st);
   PrintBitArray(bs);

   int it = rand.Next();
   BitArray bi = new BitArray(32);
   for (int i = 31; i >=0; i--)
      if (((1 << i) &  it) != 0)
         bi.Set(i, true);
      else
         bi.Set(i, false);
   System.Console.WriteLine("int value: " + it);
   PrintBitArray(bi);

   // Test BitArrays that grow:
   BitArray b127 = new BitArray(64);
   //! Would throw ArgumentOutOfRangeException
   //! b127.Set(127, true);
   //Must manually expand the Length
   b127.Length = 128;
   b127.Set(127, true);
```

```
      System.Console.WriteLine(
         "set bit 127: " + b127);
   }
} ///:~
```

Dictionaries

Dictionaries allow you to rapidly look up a value based on a unique non-numeric key and are among the most handy of the collection classes.

Hashtable

The **Hashtable** is so commonly used that many programmers use the phrase interchangeably with the concept of a dictionary! The Hashtable, though, is an implementation of **IDictionary** that has all types of interesting implementation details. Before we get to those, here's a simple example of using a **Hashtable**:

```
//:c09:SimpleHash.cs
using System.Collections;

public class LarrysPets {
    static IDictionary Fill(IDictionary d) {
        d.Add("dog", "Cheyenne");
        // Non-unique key causes exception
        //! d.Add("dog", "Bette");
        d.Add("cat", "Harry");
        d.Add("goldfish", null);
        return d;
    }
    public static void Main() {
        IDictionary pets = new Hashtable();
        Fill(pets);
        foreach(DictionaryEntry pet in pets){
            System.Console.WriteLine(
                "Larry has a {0} named {1}",
                pet.Key, pet.Value);
        }
    }
} ///:~
```

produces output of:

```
Larry has a dog named Cheyenne
Larry has a goldfish named
Larry has a cat named Harry
```

Note that attempting to add a non-unique key to a **Hashtable** raises an **ArgumentException.** This does not mean that one cannot change the value of a Hashtable at a given key, though:

```
//:c09:ChangeHashtableValue.cs
using System;
using System.Collections;

class ChangeHashtableValue{
    public static void Main(){
        Hashtable h = new Hashtable();
        h.Add("Foo","Bar");
        Object o = h["Foo"];
        h["Foo"] = "Modified";
        System.Console.WriteLine("Value is: " +
h["Foo"]);
        h["Baz"] = "Bozo";
    }
}///:~
```

This example shows the use of C#'s *custom indexers*. A custom indexer allows one to access an **IDictionary** using normal array notation. Although here we use only **string**s as the keys, the keys in an **IDictionary** can be of any type and can be mixed and match as necessary.

After "Foo" is set as the key to the "Bar" value, array notation can be used to access the value, for both reading *and* writing. As shown in the last line of **Main()**, the same array notation can be used to add new key-value pairs to the Hashtable.

The most interesting Hashtable implementation detail has to do with the calculation of the hashcode, a unique integer which "somehow" identifies the key's unique value. The hashcode is returned by **object.GetHashCode()**, a method that needs to be fast and to return integers that are "spread out" as much as possible. Additionally, the method must always return the same value for a given object, so you can't

base your hashcode on things like system time. In this example, the hashcode and the related **object.Equals()** method are used to express the idea that the sole determinant of a circle's identity is its center and radius :

```
//:c09:OverridingHash.cs
using System;
using System.Collections;
class Circle{
    int x, y, radius;
    internal Circle(int x, int y, int radius){
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    internal Circle(int topX, int topY, int lowerX,
      int lowerY){
        this.x =  topX + (lowerX - topX) / 2;
        this.y = topY + (lowerY - topY) / 2;
        this.radius = (lowerX - topX) / 2;
    }

    public override int GetHashCode(){
        System.Console.WriteLine(
            "Returning {0}", x + y + radius);
        return x + y + radius;
    }

    public override bool Equals(Object o){
        if(o is Circle){
            Circle that = (Circle) o;
            System.Console.WriteLine(
                "Comparing {0},{1},{2} with " +
                "{3},{4},{5}", x, y, radius,
                that.x, that.y, that.radius);
            return (this.x == that.x) &&
                (this.y == that.y) &&
                (this.radius == that.radius);
        }
        return false;
```

```
    }

    public static void Main(){
        Circle c = new Circle(15, 15, 5);
        Circle unlike = new Circle(15, 15, 6);
        Circle somewhatLike = new Circle(30, 1, 4);
        IDictionary d = new Hashtable();
        d.Add(c, "A circle");
        d.Add(unlike, "Another circle");
        try{
            Circle like = new Circle(10, 10, 20, 20);
            d.Add(like, "Just like c");
        }catch(Exception ex){
            System.Console.WriteLine(ex);
        }
    }
}///:~
```

When a **Circle** is added to a **Hashtable**, the Hashtable calls back to **Circle.GetHashCode()** which returns the sum of the center coordinates and radius of the circle. This is no problem for the first two circles, **c** and **unlike**, because they have different hashcodes. Circle **somewhatLike** though, causes what is called a "hash collision" – the same hashcode is returned for two different objects (in this case, both circle's elements add up to 35). When a hash collision takes place, **Hashtable** causes **object.Equals()** to see if the objects are, in fact, the same object. Because these two circles have different centers and radii, they can both be added to the **Hashtable**. Hash collisions seriously interfere with the efficiency of the **Hashtable**, so frequent collisions should make you revisit your hashcode algorithm.

In the try block, we create another new Circle, this time using an alternate constructor. When it's added to the **Hashtable**, this time there's another collision, but this time **Circle.Equals()** reveals that yes, **c** and **like** are logically equivalent and therefore **Hashtable** throws an **ArgumentException**.

ListDictionary

If you have only a dozen or fewer objects to store in your Dictionary, **ListDictionary** will have better performance than a **Hashtable**. On the

other hand, on larger amounts of objects, **ListDictionary** has much, *much* worse performance and the performance of a collection class with a small number of elements is unlikely to be a hotspot in an application. It's not impossible, though! So if you've got a dictionary with a small amount of objects and it's buried in the central loop in your application, **ListDictionary** might come in handy. Otherwise, go with **Hashtable**.

SortedList

Sometimes, you need to access a Collection in two different ways: key-based lookup for one purpose, and index-based lookup for another. The SortedList provides this dual-mode capability:

```
//:c09:ShowSortedList.cs
using System;
using System.Collections;

class ShowSortedList{
    SortedList monthList = new SortedList();
    ShowSortedList(){
        monthList.Add("January", 31);
        monthList.Add("February",28.25);
        monthList.Add("March", 31);
        monthList.Add("April", 30);
        monthList.Add("May", 31);
        monthList.Add("June", 30);
        monthList.Add("July", 31);
        monthList.Add("August", 31);
        monthList.Add("September",30);
        monthList.Add("October",31);
        monthList.Add("November",30);
        monthList.Add("December",31);

        System.Console.WriteLine(
            "June has {0} days", monthList["June"]);
        System.Console.WriteLine(
            "The eighth month has {0} days",
               monthList.GetByIndex(7));
    }

    public static void Main(){
```

```
        ShowSortedList ssl = new ShowSortedList();
    }
}///:~
```

The **SortedList** can be accessed using any object of the key type, in this case **string**s. Or, **GetByIndex**() can be used to retrieve a value based on a numeric index.

String specialists

Strings are certainly the most used type for keys and values, and the .NET Framework provides a number of specialized collections that work exclusively with strings. These collections can be found in the **System.Collections.Specialized** namespace.

One Key, Multiple Values

The **NameValueCollection** serves for those situations when you want to associate a single key **string** with multiple **string** values:

```
//:c09:Months.cs
using System;
using System.Collections.Specialized;

class Months{
    public static void Main(){
        NameValueCollection months =
           new NameValueCollection();
        months.Add("Winter", "January");
        months.Add("Winter", "February");
        months.Add("winter", "December");
        months.Add("Spring", "March");
        months.Add("Spring", "April");
        months.Add("Spring", "May");
        foreach(string key in months.AllKeys){
            System.Console.WriteLine("Key: " + key);
            System.Console.WriteLine("CSV: " +
                months[key]);
            foreach(Object value in
                months.GetValues(key)){
                System.Console.WriteLine(
                    "\tValue: " + value);
```

```
            }
        }
    }
}///:~
```

The output of the program is shown here:

```
Key: Winter
CSV: January,February,December
        Value: January
        Value: February
        Value: December
Key: Spring
CSV: March,April,May
        Value: March
        Value: April
        Value: May
```

In a rather strange design decision, the custom indexer and the **Get()** method return the values as a single comma-separated string rather than as an array. If you want to access the values as a string array, you have to use the **GetValues()** method.

Customizing Hashcode Providers

Note that in the previous Months program, the "December" value was added for the key "winter" as opposed to "January" and "February" which used the key "Winter". In the discussion of **Hashtable**, we showed how a class could override its **GetHashCode()** and **Equals()** methods to control placement in a **Hashtable**. Even more customization is possible by changing the strategy that the **Hashtable** or **NameValueCollection** uses to calculate equality; this can be done by creating the dictionary with a custom **IHashCodeProvider** and **IComparer**. By default, **NameValueCollection** uses a **CaseInsensitiveHashcodeProvider** and **CaseInsensitiveComparer** to determine what fits into what slot.

This program demonstrates the creation of a custom **IComparer** and **IHashCodeProvider** to create a hashtable which stores only the last even or odd integer added (note that this is certainly "the most complicated thing that could possibly work"):

```
//:c09:EvenOdd.cs
using System;
using System.Collections;

class EvenOddComparer : IComparer{
    public int Compare(Object x, Object y){
        //Only compare integers
        if   (x is Int32 == false
           || y is Int32 == false){
            throw new ArgumentException(
               "Can't compare non-Int32's");
        }
        //Unbox inputs
        int xValue = (int) x;
        int yValue = (int) y;
        if (xValue % 2 == yValue % 2) {
            return 0;
        }
        return -1;
    }
}

class EvenOddHashCodeProvider : IHashCodeProvider{
    public int GetHashCode(Object intObj){
        //Only hash integers
        if(intObj is Int32 == false){
            throw new ArgumentException(
               "Can't hash non-Int32's");
        }
        //Unbox input
        int x = (int) intObj;
        return x % 2;
    }
}

class EvenOdd{
    static EvenOddComparer c =
       new EvenOddComparer();
    static EvenOddHashCodeProvider p =
       new EvenOddHashCodeProvider();
    //Hashtable keys
```

```
    static readonly int EVEN_KEY = 2;
    static readonly int ODD_KEY = 3;
    //Custom IComparer & IHashCodeProvider strategies
    Hashtable evenOdd = new Hashtable(p, c);

    public void Test(){
        evenOdd[EVEN_KEY] = 2;
        evenOdd[ODD_KEY] = 3;
        evenOdd[EVEN_KEY] = 4;

        System.Console.WriteLine(
           "The last even number added was: " +
           evenOdd[EVEN_KEY]);
        System.Console.WriteLine(
           "The last odd number added was: " +
           evenOdd[ODD_KEY]);
    }

    public static void Main(){
       EvenOdd eo = new EvenOdd();
       eo.Test();
    }
}
```

String specialists: StringCollection and StringDictionary

If you only want to store strings, **StringCollection** and **StringDictionary** are marginally more efficient than their generic counterparts. A **StringCollection** implements **IList** and **StringDictionary** naturally implements **IDictionary**. Both the keys and values in **StringDictionary** must be strings, and the keys are case-insensitive and stored in lower-case form. Here's a dramatically abridged dictionary program:

```
//:c09:WebstersAbridged.cs
```

```
using System;
using System.Collections.Specialized;

class WebstersAbridged{
    static StringDictionary sd =
       new StringDictionary();

    static WebstersAbridged(){
        sd["aam"] =
          "A measure of liquids among the Dutch";
        sd["zythum"] =
          "Malt beverage brewed by ancient Egyptians";
    }

    public static void Main(string[] args){
        foreach(string arg in args){
            if(sd.ContainsKey(arg)){
                System.Console.WriteLine(sd[arg]);
            }else{
                System.Console.WriteLine(
                   "I don't know that word");
            }
        }
    }
}
```

The program iterates over the command-line arguments and either returns the definition or admits defeat. Because the **StringDictionary** is case-insensitive, this program is highly useful even when the CAPS LOCK key on the keyboard is left turned on.

Container disadvantage: unknown type

Aside from **StringCollection** and **StringDictionary**, .NET's collection classes have the "disadvantage" of obscuring type information when you put an object into a container. This happens because the programmer of that container class had no idea what specific type you wanted to put in the container, and making the container hold only your type would

prevent it from being a general-purpose tool. So instead, the container holds references to **object**, which is the root of all the classes so it holds any type. This is a great solution, except:

1. Since the type information is obscured when you put an object reference into a container, there's no restriction on the type of object that can be put into your container, even if you mean it to hold only, say, cats. Someone could just as easily put a dog into the container.

2. Since the type information is obscured, the only thing the container knows that it holds is a reference to an **object**. You must perform a cast to the correct type before you use it.

On the up side, C# won't let you *misuse* the objects that you put into a container. If you throw a dog into a container of cats and then try to treat everything in the container as a cat, you'll get a run-time exception when you pull the dog reference out of the cat container and try to cast it to a cat.

Here's an example using the basic workhorse container, **ArrayList**. First, **Cat** and **Dog** classes are created:

```
//: c09:Cat.cs
namespace pets{
    public class Cat {
        private int catNumber;
        internal Cat(int i) { catNumber = i;}
        internal void Print() {
            System.Console.WriteLine(
               "Cat #" + catNumber);
        }
    }
} ///:~

//: c09:Dog.cs
namespace pets{
    public class Dog {
        private int dogNumber;
        internal Dog(int i) { dogNumber = i;}
        internal void Print() {
            System.Console.WriteLine(
```

```
                "Dog #" + dogNumber);
        }
    }
} ///:~
```

Cats and **Dog**s are placed into the container, then pulled out:

```
//: c09:CatsAndDogs.cs
// Compile with: csc Cat.cs Dog.cs CatsAndDogs.cs
// Simple container example.
using System;
using System.Collections;

namespace pets{
    public class CatsAndDogs {
        public static void Main() {
            ArrayList cats = new ArrayList();
            for (int i = 0; i < 7; i++)
                cats.Add(new Cat(i));
            // Not a problem to add a dog to cats:
            cats.Add(new Dog(7));
            for (int i = 0; i < cats.Count; i++)
                ((Cat)cats[i]).Print();
            // Dog is detected only at run-time
        }
    }
}///:~
```

The classes **Cat** and **Dog** are distinct—they have nothing in common except that they are **object**s. (If you don't explicitly say what class you're inheriting from, you automatically inherit from **object**.) Since **ArrayList** holds **object**s, you can not only put **Cat** objects into this container using the **ArrayList** method **Add( )**, but you can also add **Dog** objects without complaint at either compile-time or run-time. When you go to fetch out what you think are **Cat** objects using the **ArrayList** indexer, you get back a reference to an **object**. Since the intent was that **cats** should only contain felines, there is no check made before casting the returned value to a **Cat**. Since we want to call the **Print()** method of **Cat**, we have to force the evaluation of the cast to happen first, so we surround the expression in parentheses before calling **Print( )**. At run-

time, though, when the loop tries to cast the **Dog** object to a **Cat**, it throws a **ClassCastException**.

This is more than just an annoyance. It's something that can create difficult-to-find bugs. If one part (or several parts) of a program inserts objects into a container, and you discover only in a separate part of the program through an exception that a bad object was placed in the container, then you must find out where the bad insert occurred. On the upside, it's convenient to start with some standardized container classes for programming, despite the scarcity and awkwardness.

Using **CollectionBase** to make type-conscious collections

As mentioned at the beginning of the chapter, the .NET runtime will eventually natively support typed collections. In the meantime, there's **CollectionBase**, an abstract **IList** that can be used as the basis for writing a strongly typed collection. To implement a typed **IList**, one starts by creating a new type that inherits from CollectionBase. Then, one has to implement **ICollection.CopyTo(), IList.Add(), IList.Contains(), IList.IndexOf(), IList.Insert(),** and **IList.Remove()** with type-specific signatures. The code is straightforward; the **CollectionBase.List** property is initialized in the base-class constructor and all your code has to do is pass your strongly-typed arguments on to **CollectionBase.List**s **object**-accepting methods and casting the **List**s **object**-returning methods to be more strongly typed:

```
//:c09:CatList.cs
//An IList that contains only Cats

using System;
using System.Collections;

namespace pets{

    class CatList : CollectionBase {
        public Cat this[int index]{
            get{ return(Cat) List[index];}
            set{ List[index] = value;}
```

```
        }

        public int Add(Cat feline){
            return List.Add(feline);
        }

        public void Insert(int index, Cat feline){
            List.Insert(index, feline);
        }

        public int IndexOf(Cat feline){
            return List.IndexOf(feline);
        }

        public bool Contains(Cat feline){
            return List.Contains(feline);
        }

        public void Remove(Cat feline){
            List.Remove(feline);
        }

        public void CopyTo(Cat[] array, int index){
            List.CopyTo(array, index);
        }

        public static void Main(){
            CatList cl = new CatList();
            for (int i = 0; i < 3; i++) {
                cl.Add(new Cat(i));
            }
            //! Can't Add(dog);
            //! cl.Add(new Dog(4));
        }
    }
}///:~
```

Note that if **CatList** had inherited directly from **ArrayList**, the methods that take references to **Cat**s, such as **Add(Cat)** would simply overload (not override) the **object**-accepting methods (e.g., **Add(object)** would still be available). Thus, the **CatList** becomes a *surrogate* to the **ArrayList**, performing some activities before passing on the

responsibility (see *Thinking in Patterns with Java*, downloadable at *www.BruceEckel.com*).

Because a **CatList** will accept only a **Cat**, the line:

```
cl.add(new Dog(4));
```

will generate an error message *at compile-time*. This approach, while more tedious from a coding standpoint, will tell you immediately if you're using a type improperly.

Note that no cast is necessary when using **Get( )** or the custom indexer — it's always a **Cat**.

IEnumerators

In any container class, you must have a way to put things in and a way to get things out. After all, that's the primary job of a container—to hold things. In the **ArrayList**, **Add( )** and **Get( )** are one set of ways to insert and retrieve objects. **ArrayList** is quite flexible—you can select anything at any time, and select multiple elements at once using different indexes.

If you want to start thinking at a higher level, there's a drawback: you need to know the exact type of the container in order to use it. This might not seem bad at first, but what if you start out using an **ArrayList**, and later on in your program you decide that because of the way you are using the container, you'd like to switch your code to use a typed collection descending from **CollectionBase**? Or suppose you'd like to write a piece of generic code that doesn't know or care what type of container it's working with, so that it could be used on different types of containers without rewriting that code?

The concept of an *enumerator* (or *iterator*) can be used to achieve this abstraction. An enumerator is an object whose job is to move through a sequence of objects and select each object in that sequence without the client programmer knowing or caring about the underlying structure of that sequence. In addition, an enumerator is usually what's called a "light-weight" object: one that's cheap to create. For that reason, you'll often

find seemingly strange constraints for enumerators; for example, some iterators can move in only one direction.

The .NET **IEnumerator** interface is an example of these kinds of constraints. There's not much you can do with one except:

1. Ask a collection (or any other type that implements IEnumerable) to hand you an **IEnumerator** using a method called **GetEnumerator( )**. This **IEnumerator** will be ready to move to the first element in the sequence on your first call to its **MoveNext( )** method.
2. Get the current object in the sequence with the **Current** property.
3. Attempt to move to the next object in sequence with the **MoveNext()** method. If the enumerator has reached the end of the sequence, this method returns **false**.
4. Reset to its initial state, which is *prior* to the first element (e.g., you must call **MoveNext()** once before reading the **Current** property).

That's all. It's a simple implementation of an iterator, but still powerful. To see how it works, let's revisit the **CatsAndDogs.cs** program from earlier in this chapter. In the following modified version, we've removed the errant dog and use an **IEnumerator** to iterate over the lists contents:

```
//: c09:CatsAndDogs2.cs
// Using an explicit IEnumerator
using System;
using System.Collections;

namespace pets{
    public class CatsAndDogs {
        public static void Main() {
            ArrayList cats = new ArrayList();
            for (int i = 0; i < 7; i++)
                cats.Add(new Cat(i));
            IEnumerator e = cats.GetEnumerator();
            while(e.MoveNext() != false){
                Object c = e.Current;
                ((Cat) c).Print();
```

```
            }
        }
    }
}///:~
```

You can see that the last few lines now use an **IEnumerator** to step through the sequence instead of a **for** loop. With the **IEnumerator**, you don't need to worry about the number of elements in the container.

Behind the scenes, C#'s **foreach()** blocks do an even better job of iterating over an **IEnumerable** type, since the **foreach** attempts to cast the **object** reference returned from the enumerator to a specific type. It doesn't make a lot of sense to use an **IEnumerator** when you can use:

```
foreach(Cat c in cats){
   c.Print();
}
```

Custom Indexers

Previously, we saw how **IDictionary** types allow one to use index notation to access key-value pairs using non-numeric indices. This is done by using operator overloading to create a custom indexer.

The type-safe **CatList** collection shown in the discussion of **CollectionBase** showed a custom indexer that took a numeric index, but returned a **Cat** instead of an **object** reference. You can manipulate both index and return types in a custom indexer, just as with any other C# method.

For instance, imagine a Maze which consists of Rooms connected by Corridors in the Room's walls. In such a situation, it might make sense to have the Corridors be indexed by direction in the room:

```
//:c09:Room.cs
using System;
using System.Collections;

namespace Labyrinth{
    enum Direction {
        north, south, east, west
```

```
    };

    class Room {
        static int counter = 0;
        protected string name;
        internal string Name{
            get{ return name;}
        }

        public override string ToString(){
            return this.Name;
        }
        internal Room(){
            name = "Room #:" + counter++;
        }
        Corridor[] c = new Corridor[
          Enum.GetValues(typeof(Direction)).Length];

        public Corridor this[Direction d]{
            get { return c[(int) d];}
            set { c[(int) d] = value;}
        }
    }

    class RegenSpot : Room {
        internal RegenSpot() : base(){
            name = "Regen Spot";
        }
    }
    class PowerUp : Room {
        internal PowerUp() : base(){
            name = "Power Up";
        }
    }
}///:~
```

First, we declare a **Direction** enumeration corresponding to the cardinal points. Then we declare a Room class with an internal static counter used to give each room a unique name. The name is made into a property and is also used to override the **ToString()** method to be a little more specific as to the exact room we're dealing with.

Every **Room** has an array called **c** that will hold references to its outbound **Corridors**. Although it would be shorter to just declare this array as size 4 (since we know that's how many **Direction**s there are), instead we determine the length of **Direction** by first using the static method **Enum.GetValues(),** passing it the **Direction** enumerations type, and then using the **Length** property of the resulting array (this way, we could accommodate octagonal rooms by simply adding values such as **NorthWest** to **Direction**).

The custom indexer is next and looks like this:

```
public Corridor this[Direction d]{
   get { return c[(int) d];}
   set { c[(int) d] = value;}
}
```

The first line begins with what looks like a normal declaration of a public Property of type **Corridor**, but the **this[Type t]** that ends the line indicates that it is an customer indexer, in this case one that takes a **Direction** value as its key and returns a **Corridor**. Since enums are value types that default to being represented by integer constants that start with zero, we can safely use the cast to int to create a numeric index to the **c** array of **Corridor**s. Like a Property, a custom indexer's **set** method has a hidden parameter called **value**.

That finishes up the **Room** class, but we declare two additional types to inherit from it – a **RegenSpot** and a **PowerUp**. They each differ from the base class solely in the way they set up their **Name** property.

The **Corridor** class referenced in **Room** has one duty – maintain references to two different **Rooms** and supply a **Traverse()** function which returns whichever of the two rooms isn't passed in as an argument:

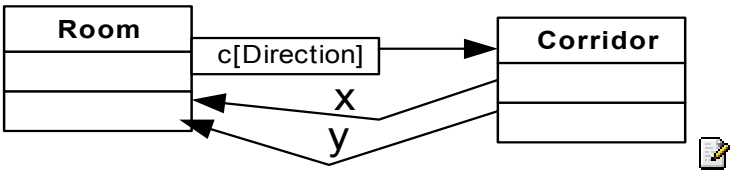
```
 //:c09:Corridor.cs
using System;
using System.Collections;

namespace Labyrinth{

    class Corridor {
        Room x, y;
```

```
        internal Corridor(
           Room x, Direction xWall,
           Room y, Direction yWall){
            this.x = x;
            this.y = y;
            x[xWall] = this;
            y[yWall] = this;
        }

        public Room Traverse(
           Room origin, Direction wall){
            System.Console.WriteLine(
               "Leaving {0} by its {1} corridor",
               origin, wall);
            if (origin == x)
                return y;
            else
                return x;
        }

        public Room Traverse(Room origin){
            System.Console.WriteLine(
               "Retreating from " + origin);
            if (origin == x)
                return y;
            else
                return x;
        }
    }
}///:~
```

The **Corridor()** constructor uses **Room**'s custom indexers (e.g., **x[xWall] = this;**). Note that there's no problem in referring to **this** inside a constructor.

Custom Enumerators & Data Structures

In the previous example, after a corridor is created, the **Corridor** contains a reference to both **Room**s, and both **Room**s contain a reference to the **Corridor**:



While we called them rooms and corridors, what we've really got here is a *non-directed graph*. Here, "graph" is being used in its mathematical sense of "a set V of Vertices and a set E of Edges that connect elements in V." What we've designed in "non-directed" because our **Corridor**s can be traversed in either direction.

A lot of very interesting problems can be mapped into graph theory (off the top of the head, problems such as: winning a game of chess, how best to pack a container, the most efficient way to schedule a bunch of jobs, and everyone's favorite, which is the cheapest route for a traveling salesperson to visit a bunch of cities). Writing a custom enumerator (or perhaps more than one, to try out different algorithms) is an elegant way to traverse a complex graph.

In this example, we create a simple Maze that consists of one **RegenSpot** and one **PowerUp**, and several normal rooms (the names are taken from videogames for which one can program "bots" – just think of them as the start and stop points):

```
//:c09:Maze.cs
using System;
using System.Collections;
namespace Labyrinth{
    class Maze :IEnumerable {
        Room[] rooms;
        Room regenRoom;
```

```
        internal Room RegenSpot{
            get { return regenRoom;}
        }

        Maze(){
            regenRoom = new RegenSpot();
            rooms = new Room[]{
                new Room(), new Room(), new Room(),
                regenRoom, new Room(), new Room(),
                new PowerUp()
            };
            new Corridor(rooms[0], Direction.east,
               rooms[4], Direction.north);
            new Corridor(rooms[0], Direction.south,
               rooms[1], Direction.north);
            new Corridor(rooms[1], Direction.south,
               rooms[3], Direction.north);
            new Corridor(rooms[2], Direction.east,
               rooms[3], Direction.west);
            new Corridor(rooms[3], Direction.east,
               rooms[4], Direction.west);
            new Corridor(rooms[3], Direction.south,
               rooms[5], Direction.south);
            new Corridor(rooms[5], Direction.south,
               rooms[6], Direction.north);
        }

        public static void Main(){
            Maze m = new Maze();
            foreach(Room r in m){
                System.Console.WriteLine(
                   "RoomRunner in " + r.Name);
            }
        }

        public IEnumerator GetEnumerator(){
            return new DepthFirst(this);
        }
    }
}///:~
```

Class **Maze** is declared to implement the **IEnumerable** interface, which we'll use to return a customized enumerator which runs the maze. Note that for our purposes, we don't care if the enumerator visits every vertex on the graph (every room in the maze); as a matter of fact, we're probably most interested in an enumerator which visits as few vertices as possible! This is a different intention from the generic enumerators of the .NET collection classes, which of course *do* need to visit every element in the data structure.

The **Maze** contains an array **rooms** and a reference to the starting **RegenRoom**. The maze's dynamic structure is built in the **Maze()** constructor and consists of 7 **Room**s and 7 **Corridor**s.

The **Main()** method constructs a **Maze** and then uses a **foreach** block to show the traversal of the maze. Behind the scenes, the **foreach** block determines that **Maze** is an **IEnumerable** type and silently calls **GetEnumerator()**.

Maze's implementation of **IEnumerable.GetEnumerator()** is the final method in **Maze**. A new object of type **DepthFirst** (discussed shortly) is created with a reference to the current **Maze**.

There are several different ways to traverse a maze. It is probable that when writing a program to run mazes, you would want to try several different algorithms, one that rushed headlong down the first unexplored corridor, another that methodically explored all the routes from a single room, etc. However, each of these algorithms has a lot of things in common: they must all implement the **IEnumerator** interface, they all have references to the **Maze** and a current **Room**, they all begin at the regen spot and end at the power up. Really, the only way they differ is in their implementation of **IEnumerator.MoveNext()** when they're "lost" in the maze. This is a job for the "Template Method" pattern:

```
//:c09:RoomRunner.cs
using System;
using System.Collections;
namespace Labyrinth{
    abstract class RoomRunner:IEnumerator {
        Maze m;
        protected RoomRunner(Maze m){
```

```
            this.m = m;
        }

        protected Room currentRoom = null;
        public Object Current{
            get { return currentRoom;}
        }


        public virtual void Reset(){
            currentRoom = null;
        }

        public bool MoveNext(){
            if (currentRoom == null) {
                System.Console.WriteLine(
                 "{0} starting the maze",
                 this.GetType());
                currentRoom = m.RegenSpot;
                return true;
            }
            if (currentRoom is PowerUp) {
                System.Console.WriteLine(
                   "{0} has found PowerUp!",
                   this.GetType());
                return false;
            }
            return this.ConcreteMoveNext();
        }

        protected abstract bool ConcreteMoveNext();
    }
}///:~
```

Here, an **abstract** class called **RoomRunner** implements the methods of the **IEnumerator** interface, but leaves one tiny bit to its subclasses to implement.

The RoomRunner() constructor just stores a reference to the **Maze** that creates it and initializes the **currentRoom** (exposed to the outside world as **IEnumerator**'s **Current** Property) to null. **Reset()** also sets the

currentRoom to null – remember that **IEnumerator.MoveNext()** is always called once *before* the first read of the **Current** property.

The first time **RoomRunner.MoveNext()** is called, **currentRoom** will be null. Because **RoomRunner** is an abstract type that may be implemented by many different subtypes, our console message can use **this.GetType()** to determine the exact runtime type of **RoomRunner**. (The trick at the root of the "template method" pattern.) After printing a message to the screen announcing the **RoomRunner**'s readiness to start traversing the **Maze**, the current room is set to the **Maze**'s **RegenSpot** and the method returns true to indicate that the **IEnumerator** is at the beginning of the data structure.

Similarly, if the **currentRoom** is of type **PowerUp**, the maze running is, by our definition, complete and **MoveNext()** returns false.

If, however, the currentRoom is neither null nor a PowerUp room, execution goes to **this.ConcreteMoveNext()**. This is the template method. Just as **this.GetType()** will return the exact runtime type, **this.ConcreteMoveNext()** will execute the **ConcreteMoveNext()** method of the runtime type. For this to work, of course, **RoomRunner.ConcreteMoveNext()** must be declared as **virtual** or, as in this case, **abstract**.

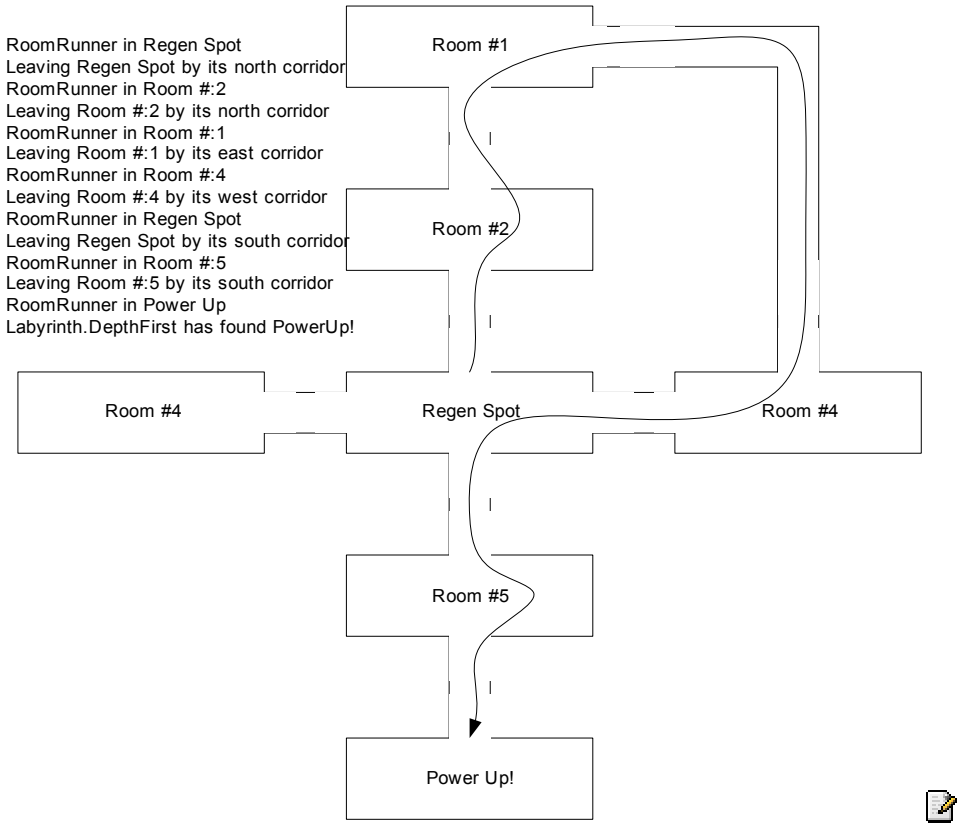
Maze.GetEnumerator() returned an object of type **DepthFirst**, which implements **RoomRunner**'s template method **ConcreteMoveNext()**:

```
//:c09:DepthFirst.cs
using System;
using System.Collections;
namespace Labyrinth{
    class DepthFirst : RoomRunner {
        public DepthFirst(Maze m) : base(m){}

        ArrayList visitedCorridors = new ArrayList();
        Corridor lastCorridor = null;

        protected override bool ConcreteMoveNext(){
            foreach(Direction d in
               Enum.GetValues(typeof(Direction))){
```

```
                if (currentRoom[d] != null) {
                   Corridor c = currentRoom[d];
                   if (visitedCorridors.Contains(c)
                       == false) {
                       visitedCorridors.Add(c);
                       lastCorridor = c;
                       currentRoom =
                         c.Traverse(currentRoom, d);
                       return true;
                   }
                }
            }
            //No unvisited corridors! Retreat!
            lastCorridor.Traverse(currentRoom);
            return true;
        }
    }
}///:~
```

DepthFirst inherits from BaseRunner, as its class declaration and constructor show. This particular maze runner essentially goes down the first corridor it sees that it hasn't yet gone through. If there are none it hasn't been down, it backtracks to the previous room. Uh-oh – what if it backtracks into a room and *that* room doesn't have any unvisited corridors? Looks like a defect!

But on our maze, the DepthFirst works like a champ:


RoomRunner in Regen Spot
Leaving Regen Spot by its north corridor
RoomRunner in Room #:2
Leaving Room #:2 by its north corridor
RoomRunner in Room #:1
Leaving Room #:1 by its east corridor
RoomRunner in Room #:4
Leaving Room #:4 by its west corridor
RoomRunner in Regen Spot
Leaving Regen Spot by its south corridor
RoomRunner in Room #:5
Leaving Room #:5 by its south corridor
RoomRunner in Power Up
Labyrinth.DepthFirst has found PowerUp!
Room #1
Room #2
Room #4
Regen Spot
Room #4
Room #5
Power Up!

Sorting and searching **List**s

Utility methods sort and search in **List**s have the same names and signatures as those for arrays of objects, but are instance methods of **IList** instead of **Array**. Here's an example:

```
//: c09:ListSortSearch.cs
// Sorting and searching ILists
using System;
using System.Collections;

public class ListSortSearch {
    private static void Fill(ArrayList list){
```

```
        for (int i = 0; i < 25; i++) {
            char c = (char) ('A' + i);
            list.Add(c);
        }
    }

    private static void Print(ArrayList list){
        foreach(Object o in list){
            System.Console.Write(o + ", ");
        }
        System.Console.WriteLine();
    }

    private static void Shuffle(ArrayList list){
        int len = list.Count;
        for (int i = 0; i < len; i++) {
            int k = rand.Next(len);
            Object temp = list[i];
            list[i] = list[k];
            list[k] = temp;
        }
    }

    static Random rand = new Random();

    public static void Main() {
        ArrayList list = new ArrayList();
        Fill(list);
        Print(list);
        Shuffle(list);
        System.Console.WriteLine("After shuffling: ");
        Print(list);
        list.Reverse();
        System.Console.WriteLine("Reversed: ");
        Print(list);
        list.Sort();
        System.Console.WriteLine("After sorting: ");
        Print(list);
        Object key = list[12];
        int index = list.BinarySearch(key);
        System.Console.WriteLine(
```

```
            "Location of {0} is {1}, list[{2}] = {3}",
            key, index, index, list[index]);
    }
} ///:~
```

The use of these methods is identical to the **static** ones in **Array**, but are instance methods of **ArrayList**. This program also contains an implementation of Donald Knuth's shuffling algorithm to randomize the order of a **List**.

From Collections to Arrays

The ICollection interface specifies that all collections must be able to copy their contents into an array. The destination array must be a one-dimensional array with zero-based indexing. The copying procedure may insert objects into the array starting at an arbitrary index, assuming of course that the index is zero or positive and that the destination array is properly initialized.

Copying an ArrayList to an array is simple:

```
//:c09:ListToArray.cs
using System;
using System.Collections;

class ListToArray{
    public static void Main(){
        IList list = new ArrayList();
        Random r = new Random();
        int iToAdd = 10 + r.Next(10);
        for(int i = 0; i < iToAdd; i++){
            list.Add(r.Next(100));
        }
        int indexForCopyStart = r.Next(10);
        int[] array = new int[
           indexForCopyStart + list.Count];
        list.CopyTo(array, indexForCopyStart);
        for(int i = 0; i < array.Length; i++){
           System.Console.WriteLine(
              "array[{0}] = {1}", i, array[i]);
```

```
        }
    }
}///:~
```

After initializing an **ArrayList**, we use a random number generator to choose to add between 10 and 19 items. We loop, using **list.Add()** to add random numbers between 0 and 99. Then, we choose a random number to indicate where in the array we wish to begin copying. We then declare and initialize the array, which must be sized to accommodate **indexForCopyStart** empty integers (since it's an array of ints, these will be initialized to 0) and **list.Count** integers from the **ArrayList**. The **CopyTo()** method takes two parameters – the reference to the destination array and the starting index for the copy. We then loop over the array, outputting the contents.

Since integers are value types, modifying values in the destination **array** will not be reflected in the **ArrayList list**. However, reference types would naturally have the same "shallow copy" issues that are discussed in #ref to shallow copy discussion.#

Since **IDictionary** inherits from **ICollection**, implementing types must support **CopyTo()**. The results are an array of **DictionaryEntry** items:

```
//:c09:DictionaryToArray.cs
using System;
using System.Collections;

class DictionaryToArray{
   public static void Main(){
      IDictionary dict = new Hashtable();
      Random r = new Random();
      int iKeys = 3 + r.Next(3);
      for(int i = 0; i < iKeys; i++){
         dict.Add(i, r.Next(100));
      }
      DictionaryEntry[] a =
         new DictionaryEntry[dict.Count];
      dict.CopyTo(a, 0);
      for(int i = 0; i < a.Length; i++){
         DictionaryEntry de = a[i];
         System.Console.WriteLine(
```

```
            "a[{0}]: .Key = {1} .Value = {2}",
            i, de.Key, de.Value);
      }
   }
}///:~
```

A typical run looks like:

```
a[0]: .Key = 4 .Value = 11
a[1]: .Key = 3 .Value = 5
a[2]: .Key = 2 .Value = 6
a[3]: .Key = 1 .Value = 93
a[4]: .Key = 0 .Value = 4
```

You'll note that the resulting array is not sorted on the value of the key, which might be desirable, because **IDictionary** doesn't require keys to be **IComparable**. However, they often are and, if so, it would probably be nice if the resulting array were ordered by key. This program demonstrates a technique to get this result:

```
//:c09:DictionaryToSortedArray.cs
using System;
using System.Collections;

class DictionaryToArray{
   public static void Main(){
      IDictionary dict = new Hashtable();
      Random r = new Random();
      int iKeys = 3 + r.Next(3);
      for(int i = 0; i < iKeys; i++){
         dict.Add(i, i);
      }

      //First, get array of keys
      ICollection keyCol = dict.Keys;
      IComparable[] keyArray =
         new IComparable[keyCol.Count];
      //Would throw exception if keys not IComparable
      keyCol.CopyTo(keyArray, 0);

      //Second, get array of values
      ICollection valCol = dict.Values;
      Object[] valArray = new Object[valCol.Count];
```

```
      valCol.CopyTo(valArray, 0);

      Array.Sort(keyArray, valArray);

      //Instantiate destination array
      DictionaryEntry[] a =
         new DictionaryEntry[keyCol.Count];
      //Retrieve and set in key-sorted order
      for(int i = 0; i < a.Length; i++){
         a[i] = new DictionaryEntry();
         a[i].Key = keyArray[i];
         a[i].Value = valArray[i];
      }

      //Output results
      for(int i = 0; i < a.Length; i++){
         DictionaryEntry de = a[i];
         System.Console.WriteLine(
            "a[{0}]: .Key = {1} .Value = {2}",
            i, de.Key, de.Value);
      }
   }
}///:~
```

The program starts off similarly to the previous examples, with a few key-value pairs being inserted into a **Hashtable**. Instead of directly copying to the destination array, though, we retrieve the **ICollection** of keys. **ICollection** doesn't have any sorting capabilities, so we use **CopyTo()** to move the keys into an **IComparable[]** array. If any of our keys did *not* implement **IComparable**, this would throw an **InvalidCastException**.

The next step is to copy the values from the **Hashtable** into another array, this time of type **object[]**. We then use **Array.Sort(Array, Array)**, which sorts *both* its input arrays based on the comparisons in the first array, which in our case is the key array. In general, one should avoid a situation where one changes the state of one object (such as the array of values) based on logic internal to another object (such as the sorting of the array of keys); a situation that's called *logical cohesion*. We could avoid using **Array.Sort(Array, Array)** by sorting **keyArray** and then using a **foreach(object key in keyArray)** loop to retrieve the values

from the Hashtable, but in this case that's closing the barn door after the horses have fled – the .NET Framework does not have an **IDictionary** which maintains its objects in key order, which would be the best solution for the general desire to move between an **IDictionary** and a sorted array.

In keeping with its singular nature, **NameValueCollection.CopyTo()** does not act like **Hashtable.CopyTo()**. Where **Hashtable.CopyTo()** creates an array of **DictionaryEntry** objects that contain both the key and value, **NameValueCollection**'s **CopyTo()** method creates an array of **string**s which represent the values in a comma-separated format, with no reference to the keys. It's difficult to imagine the utility of this format.

This example builds a more reasonable array from a NameValueCollection:

```
//:c09:NValColToArray.cs
using System;
using System.Collections;
using System.Collections.Specialized;

class NameValueCollectionEntry : IComparable{
   string key;
   public string Key{
      get { return key; }
   }

   string[] values;
   public string[] Values{
      get { return values; }
   }

   public NameValueCollectionEntry(
      string key, string[] values){
      this.key = key;
      this.values = values;
   }

   public int CompareTo(Object o){
      if(o is NameValueCollectionEntry){
```

```
            NameValueCollectionEntry that =
               (NameValueCollectionEntry) o;
            return this.Key.CompareTo(that.Key);
         }
         throw new ArgumentException();
      }

      public static NameValueCollectionEntry[]
        FromNameValueCollection(NameValueCollection src){
         string[] keys = src.AllKeys;
         NameValueCollectionEntry[] results =
            new NameValueCollectionEntry[keys.Length];
         for(int i = 0; i < keys.Length; i++){
            string key = keys[i];
            string[] vals = src.GetValues(key);
            NameValueCollectionEntry entry =
               new NameValueCollectionEntry(key, vals);
            results[i] = entry;
         }
         return results;
      }
   }


   class NValColToArray{
      public static void Main(){
         NameValueCollection dict =
            new NameValueCollection();
         Random r = new Random();
         int iKeys = 3 + r.Next(3);
         for(int i = 0; i < iKeys; i++){
            int iValsToAdd = 5 + r.Next(5);
            for(int j = 0; j < iValsToAdd; j++){
               dict.Add(i.ToString(),
                        r.Next(100).ToString());
            }
         }

         NameValueCollectionEntry[] a =
           NameValueCollectionEntry.
             FromNameValueCollection(dict);
```

```
      Array.Sort(a);
      for(int i = 0; i < a.Length; i++){
         System.Console.WriteLine(
            "a[{0}].Key = {1}", i, a[i].Key);

         foreach(string v in a[i].Values){
            System.Console.WriteLine(
               "\t Value: " + v);
         }
      }
   }
}///:~
```

We define a new type, **NameValueCollectionEntry** which corresponds to the **DictionaryEntry** of **Hashtable**. Like that type, our new type has a **Key** and a **Value** property, but these are of type **string** and **string[]** respectively. Because we know that the **Key** is always going to be a **string**, we can declare **NameValueCollectionEntry** to implement **IComparable** and implement that simply by comparing **Key** properties (if the parameter to **CompareTo()** is not a **NameValueCollectionEntry**, we throw an **ArgumentException**).

The static **FromNameValueCollection()** method is where we convert a **NameValueCollection** into an array of **NameValueCollectionEntry**s. First, we get a **string[]** of keys from the **AllKeys** property of the input parameter (if we had used the **Keys** property, we would have received the same data, but in an **object** array). The **Length** propery of the **keys** allows us to size the **results** array. The **GetValues()** method of **NameValueCollection** returns a **string** array, which along with the **string** key, is what we need to instantiate a single **NameValueCollectionEntry**. This entry is added to the **results** which are returned when the loop ends.

Class **NValColToArray** demonstrates the use of this new class we've written. A **NameValueCollection** is created and each entry is filled with a random number of strings. A **NameValueCollectionEntry** array called **a** is generated using the static function just discussed. Since we implemented **IComparable** in **NameValueCollectionEntry**, we can use **Array.Sort()** to sort the results by the **Key string**s. For each **NameValueCollectionEntry**, we output the **Key,** retrieve the **string[]**

Values, and output them to the console. We could, of course, sort the **Values** as well, if that was desired.

Persistent Data With ADO.NET

While collection classes and data structures remain important to in-memory manipulation of data, offline storage is dominated by third-party vendors supporting the relational model of data storage. Of course, Oracle's eponymous product dominates the high-end market, while Microsoft's SQL Server and IBM's DB2 are able competitors for enterprise data. There are hundreds of databases appropriate for smaller projects, the most well-distributed of which is Microsoft's Access.

Meanwhile, the success of the Web made many people comfortable with the concept that "just" text-based streams marked with human-readable tags were sufficiently powerful to create a lot of end-user value. Extensible Markup Language (XML) has exploded in popularity in the new millennium and is rapidly becoming the preferred in-memory representation of relational data. This will be discussed in depth in chapter #XML#, but some discussion of XML is relevant to any discussion of Active Data Objects for .NET (ADO.NET).

Like graphics programming, the complete gamut of database programming details cannot be adequately covered in less than an entire book. However, also like graphics programming, most programmers do not need to know more than the basics. In the case of database programming, 99% of the work boils down to being able to Create, Read, Update, and Delete data – the functions known as "CRUD." This section tries to deliver the *minimum* amount of information you need to be able to use ADO.NET in a professional setting.

Although CRUD may encapsulate many programmers intent for ADO.NET, there's another "D" that is fundamental to ADO.NET – "Disconnected." Ironically, the more the World Wide Web becomes truly ubiquitous, the more difficult it is to create solutions based on continuous connections. The client software that is running in a widely distributed

application, i.e., an application that is running over the Internet, simply cannot be counted on to go through an orderly, timely lifecycle of "connect, gain exclusive access to data, conduct transactions, and disconnect." Similarly, although continuous network access may be the rule in corporate settings, the coming years are going to see an explosion in applications for mobile devices, such as handhelds and telephones, which have metered costs; economics dictate that such applications cannot be constantly connected.

ADO.NET separates the tasks of actually accessing the data store from the tasks of manipulating the resulting set of data. The former obviously require a connection to the store while the latter do not. This separations of concerns helps when converting data from one data store to another (such as converting data between relational table data and XML) as well as making it much easier to program widely-distributed database applications. However, this model increases the possibility that two users will make incompatible modifications to related data – they'll both reserve the last seat on the flight, one will mark an issue as resolved while the other will expand the scope of the investigation, etc. So even a minimal introduction to ADO.NET requires some discussion of the issues of concurrency violations.

Getting a handle on data with DataSet

The **DataSet** class is the root of a relational view of data. A **DataSet** has **DataTable**s, which have **DataColumn**s that define the types in **DataRow**s. The relational database model was introduced by E.F. Codd in the early 1970s. The concept of tables storing data in rows in strongly-typed columns may seem to be the very definition of what a database is, but Codd's formalization of these concepts and others such such as *normalization* (a process by which redundant data is eliminated and thereby ensuring the correctness and consistency of edits) was one of the great landmarks in the history of computer science.

While normally one creates a **DataSet** based on existing data, it's possible to create one from scratch, as this example shows:

```
//:c09:BasicDataSetOperations.cs
```

```
using System.Data;

class BasicDataSetOperations {
    public static void Main(string[] args){
        DataSet ds = BuildDataSet();
        PrintDataSetCharacteristics(ds);
    }

    private static DataSet BuildDataSet() {
        DataSet ds = new DataSet("MockDataSet");

        DataTable auTable = new DataTable("Authors");
        ds.Tables.Add(auTable);

        DataColumn nameCol = new DataColumn("Name",
            typeof(string));
        auTable.Columns.Add(nameCol);

        DataRow larryRow = auTable.NewRow();
        larryRow["Name"] = "Larry";
        auTable.Rows.Add(larryRow);
        DataRow bruceRow = auTable.NewRow();
        bruceRow["Name"] = "Bruce";
        auTable.Rows.Add(bruceRow);

        return ds;
    }

    private static void PrintDataSetCharacteristics(
       DataSet ds){
      System.Console.WriteLine(
        "DataSet \"{0}\" has {1} tables",
        ds.DataSetName, ds.Tables.Count);
      foreach(DataTable table in ds.Tables){
         System.Console.WriteLine(
            "Table \"{0}\" has {1} columns",
            table.TableName, table.Columns.Count);
         foreach(DataColumn col in table.Columns){
             System.Console.WriteLine(
                 "Column \"{0}\" contains data of type
{1}",
```

```
                col.ColumnName, col.DataType);
        }

        System.Console.WriteLine(
            "The table contains {0} rows",
            table.Rows.Count);
        foreach(DataRow r in table.Rows){
            System.Console.Write("Row Data: ");
            foreach(DataColumn col in table.Columns){
                   string colName = col.ColumnName;
                   System.Console.Write(
                      "[{0}] = {1}",
                      colName, r[colName]);
               }
               System.Console.WriteLine();
           }

       }
    }

}///:~
```

The .NET classes related to **DataSets** are in the **System.Data** namespace, so naturally we have to include a **using** statement at the beginning of the program. The **Main()** method is straightforward, it calls **BuildDataSet()** and passes the object returned by that method to another static method called **PrintDataSetCharacteristics()**.

BuildDataSet() introduces several new classes. First comes a **DataSet**, using a constructor that allows us to simultaneously name it "MockDataSet." Then, we declare and initialize a **DataTable** called "Author" which we reference with the **auTable** variable. **DataSet** objects have a **Tables** property of type **DataTableCollection**, which implements **ICollection**. While **DataTableCollection** does not implement **IList**, it contains some similar methods, including **Add**, which is used here to add the newly created **auTable** to **ds**'s **Tables**.

DataColumns, such as the **nameCol** instantiated in the next line, are associated with a particular **DataType**. **DataType**s are not nearly as extensive or extensible as normal types. Only the following can be specified as a **DataType**:

- Boolean
- Byte
- Char
- DateTime
- Decimal
- Double
- Int16
- Int32
- Int64
- SByte
- Single
- String
- TimeSpan
- UInt16
- UInt32
- UInt64

In this case, we specify that the "Name" column should store strings. We add the column to the **Columns** collection (a **DataColumnCollection**) of our **auTable**.

One cannot create rows of data using a standard constructor, as a row's structure must correspond to the **Columns** collection of a particular **DataTable**. Instead, **DataRow**s are constructed by using the **NewRow()** method of a particular **DataTable**. Here, **auTable.NewRow()** returns a **DataRow** appropriate to our "Author" table, with its single "Name" column. **DataRow** does not implement **ICollection**, but does overload the indexing operator, so assigning a

value to a column is as simple as saying: `larryRow["Name"] = "Larry"`.

The reference returned by **NewRow()** is *not* automatically inserted into the **DataTable** which generates it, that is done by:

```
auTable.Rows.Add(larryRow);
```

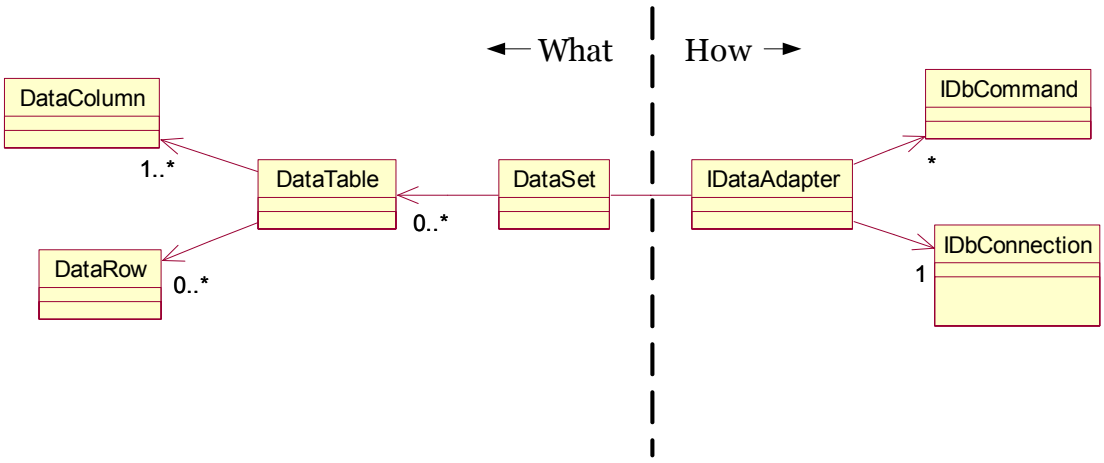
After creating another row to contain Bruce's name, the **DataSet** is returned to the **Main()** method, which promptly passes it to **PrintDataSetCharacteristics()**. The output is:

```
DataSet "MockDataSet" has 1 tables
Table "Authors" has 1 columns
Column "Name" contains data of type System.String
The table contains 2 rows
Row Data: [Name] = Larry
Row Data: [Name] = Bruce
```

Connecting to a database

The task of actually moving data in and out of a store (either a local file or a database server on the network) is the task of the **IDbConnection** interface. Specifying which data (from all the tables in the underlying database) is the responsibility of objects which implement IDbCommand. And bridging the gap between these concerns and the concerns of the **DataSet** is the responsibility of the **IDbAdapter** interface.

Thus, while **DataSet** and the classes discussed in the previous example encapsulate the "what" of the relational data, the **IDataAdapter**, **IDbCommand**, and **IDbConnection** encapsulate the "How":



The .NET Framework currently ships with two *managed providers* that **IDataAdapter** and its related classes. One is high-performance provider optimized for Microsoft SQL Server; it is located in the System.Data.SqlClient namespace. The other provider, in the System.Data.OleDb namespace, is based on the broadly available Microsoft JET engine (which ships as part of Windows XP and is downloadable from Microsoft's Website). Additionally, you can download an ODBC-suppporting managed provider from msdn.microsoft.com. One suspects that high-performance managed providers for Oracle, DB2, and other high-end databases will quietly become available as .NET begins to achieve significant market share.

For the samples in this chapter, we're going to use the OleDb classes to read and write an Access database, but we're going to upcast everything to the ADO.NET interfaces so that the code is as general as possible.

The "Northwind" database is a sample database from Microsoft that you can download from http://msdn.microsoft.com/downloads if you don't already have it on your hard-drive from installing Microsoft Access. The file is called "nwind.mdb". Unlike with enterprise databases, there is no need to run a database server to connect to and manipulate an Access database. Once you have the file you can begin manipulating it with .NET code.

This first example shows the basic steps of connecting to a database and filling a dataset:

```
//:c09:DBConnect.cs
using System.Data;
using System.Data.OleDb;

class BasicDataSetOperations {
    public static void Main(string[] args){
        DataSet ds = Employees("Nwind.mdb");
        System.Console.WriteLine(
            "DS filled with {0} rows",
            ds.Tables[0].Rows.Count);
    }
    private static DataSet Employees(
        string fileName){
        IDbConnection cnctn = new OleDbConnection();
        cnctn.ConnectionString=
            "Provider=Microsoft.JET.OLEDB.4.0;" +
            "data source=" + fileName;
        try {
          cnctn.Open();

          string selStr = "SELECT * FROM EMPLOYEES";
          IDataAdapter adapter =
            new OleDbDataAdapter(selStr, cnctn);

          DataSet ds = new DataSet("Employees");
          adapter.Fill(ds);
        } finally {
          cnctn.Close();
        }

        return ds;
    }
}///:~
```

After specifying that we'll be using the System.Data and System.Data.OleDb namespaces, the **Main()** initializes a **DataSet** with the results of a call to the static function **Employees()**. The number of rows in the first table of the result is printed to the console.

The method **Employees()** takes a string as its parameter in order to clarify the part of the connection string that is variable. In this case, you'll

obviously have to make sure that the file "Nwind.mdb" is in the current directory or modify the call appropriately.

In order to emphasize that ADO.NET provides abstract data types for database connection, after initializing an **OleDbConnection** we upcast the result to the **IDbConnection** interface.

The **ConnectionString** property is set to a bare minimum: the name of the provider we intend to use and the data source. This is all we need to connect to the Northwind database, but enterprise databases will often have significantly more complex connection strings.

The call to **cnctn.Open()** starts the actual process of connecting to the database, which in this case is a local file read but which would typically be over the network. Because database connections are the prototypical "valuable non-memory resource," as discussed in Chapter #Exceptions#, we put the code that interacts with the database inside a **try…finally** block.

As we said, the **IDataAdapter** is the bridge between the "how" of connecting to a database and the "what" of a particular relational view into that data. The bridge going from the database to the **DataSet** is the **Fill()** method (while the bridge from the **DataSet** to the database is the **Update()** method which we'll discuss in our next example). How does the **IDataAdapter** know what data to put into the **DataSet**? The answer is actually not defined at the level of **IDataAdapter**. The **OleDbAdapter** supports several possibilities, including automatically filling the **DataSet** with all, or a specified subset, of records in a given table. The **DBConnect** example shows the use of Structured Query Language (SQL), which is probably the most general solution. In this case, the SQL query `SELECT * FROM EMPLOYEES` retrieves all the columns and all the data in the EMPLOYEES table of the database.

The **OleDbDataAdapter** has a constructor which accepts a string (which it interprets as a SQL query) and an **IDbConnection**. This is the constructor we use and upcast the result to **IDataAdapter**.

Now that we have our open connection to the database and an **IDataAdapter,** we create a new **DataSet** with the name "Employees." This empty **DataSet** is passed in to the **IDataAdapter.Fill()** method,

which executes the query via the **IDbConnection**, adds to the passed-in **DataSet** the appropriate **DataTable** and **DataColumn** objects that represent the structure of the response, and then creates and adds to the **DataSet** the **DataRow** objects that represent the results.

The **IDbConnection** is **Close**d within a finally block, just in case an **Exception** was thrown sometime during the database operation. Finally, the filled **DataSet** is returned to **Main()**, which dutifully reports the number of employees in the Northwind database.

Fast Reading With an IDataReader

The preferred method to get data is to use an **IDataAdapter** to specify a view into the database and use **IDataAdapter.Fill()** to fill up a **DataSet.** An alternative, if all you want is a read-only forward read, is to use an **IDataReader**. An **IDataReader** is a direct, connected iterator of the underlying database; it's likely to be more efficient than filling a **DataSet** with an **IDataAdapter**, but the efficiency requires you to forego the benefits of a disconnected architecture. This example shows the use of an **IDataReader** on the **Employees** table of the Northwind database:

```
//:c09:DataReader.cs
using System;
using System.Data;
using System.Data.OleDb;

class DataReader {
   public static void Main(){
      EnumerateEmployees("Nwind.mdb");
   }

   private static void EnumerateEmployees(
         string fileName){
      OleDbConnection cnctn = new OleDbConnection();
      cnctn.ConnectionString=
         "Provider=Microsoft.JET.OLEDB.4.0;" +
         "data source=" + fileName;
      IDataReader rdr = null;
      try {
```

```
            cnctn.Open();
            IDbCommand sel =
            new OleDbCommand(
               "SELECT * FROM EMPLOYEES", cnctn);
            rdr = sel.ExecuteReader();
            while (rdr.Read()) {
               System.Console.WriteLine(
                  rdr["FirstName"] + " "
                + rdr["LastName"]);
            }
         } finally {
            rdr.Close();
            cnctn.Close();
         }
      }
}///:~
```

The **EnumerateEmployees()** method starts like the code in the **DBConnect** example, but we do not upcast the **OleDbConnection** to **IDbConnection** for reasons we'll discuss shortly. The connection to the database is identical, but we declare an **IDataReader rdr** and initialize it to null before opening the database connection; this is so that we can use the **finally** block to **Close()** the **IDataReader** as well as the **OleDbConnection**.

After opening the connection to the database, we create an **OleDbCommand** which we upcast to **IDbCommand**. In the case of the **OleDbCommand** constructor we use, the parameters are a SQL statement and an **OleDbConnection** (thus, our inability to upcast in the first line of the method).

The next line, `rdr = sel.ExecuteReader()`, executes the command and returns a *connected* **IDataReader**. **IDataReader.Read()** reads the next line of the query's result, returning **false** when it runs out of rows. Once all the data is read, the method enters a **finally** block, which severs the **IDataReader**'s connection with **rdr.Close()** and then closes the database connection entirely with **cnctn.Close()**.

CRUD With ADO.NET

With **DataSet**s and managed providers in hand, being able to create, read, update, and delete records in ADO.NET is near at hand. Creating data was covered in the **BasicDataSetOperations** example – use **DataTable.NewRow()** to generate an appropriate **DataRow**, fill it with your data, and use **DataTable.Rows.Add()** to insert it into the **DataSet**. Reading data is done in a flexible disconnected way with an **IDataAdapter** or in a fast but connected manner with an **IDataReader**.

Update and Delete

The world would be a much pleasanter place if data never needed to be changed or erased[2]. These two operations, especially in a disconnected mode, raise the distinct possibility that two processes will attempt to perform incompatible manipulation of the same data. There are two options for a database model:

- Assume that any read that might end in an edit *will* end in an edit, and therefore not allow anyone else to do a similar editable read. This model is known as pessimistic concurrency.
- Assume that although people will edit and delete rows, make the enforcement of consistency the responsibility of some software component other than the database components. This is optimistic concurrency, the model that ADO.NET uses.

When an **IDbAdapter** attempts to update a row that has been updated since the row was read, the second update fails and the adapter throws a **DBConcurrencyException** (note the capital 'B' that violates .NET's the naming convention).

As an example:

[2] Not only would it please the hard drive manufacturers; it would provide a way around the second law of thermodynamics.

1. Ann and Ben both read the database of seats left on the 7 AM flight to Honolulu. There are 7 seats left.
2. Ann and Ben both select the flight, and their client software shows 6 seats left.
3. Ann submits the change to the database and it completes fine.
4. Charlie reads the database, sees 6 seats available on the flight.
5. Ben submits the change to the database. Because Anne's update happened before Ben's update, Ben receives a **DBConcurrencyException**. The database does *not* accept Ben's change.
6. Charlie selects a flight and submits the change. Because the row hasn't changed since Charlie read the data, Charlie's request succeeds.

It is impossible to give even general advice as to what to do after receiving a **DBConcurrencyException**. Sometimes you'll want to take the data and re-insert it into the database as a new record, sometimes you'll discard the changes, and sometimes you'll read the new data and reconcile it with your changes. There are even times when such an exception indicates a deep logical flaw that calls for a system shutdown.

This example performs all of the CRUD operations, rereading the database after the update so that the subsequent deletion of the new record does not throw a **DBConcurrencyException**:

```
//:c09:Crud.cs
using System.Data;
using System.Data.OleDb;

class Crud {
    public static void Main(string[] args){
        Crud myCrud = new Crud();
        myCrud.ReadEmployees("NWind.mdb");
        myCrud.Create();
        myCrud.Update();
        //Necessary to avoid DBConcurrencyException
        myCrud.Reread();
        myCrud.Delete();
```

```
}

OleDbDataAdapter adapter;
DataSet emps;

private void ReadEmployees(
    string pathToAccessDB){
    OleDbConnection cnctn = new OleDbConnection();
    cnctn.ConnectionString=
        "Provider=Microsoft.JET.OLEDB.4.0;" +
        "data source=" + pathToAccessDB;
    cnctn.Open();

    string selStr = "SELECT * FROM EMPLOYEES";
    adapter = new OleDbDataAdapter(selStr, cnctn);
    new OleDbCommandBuilder(adapter);

    emps = new DataSet("Employees");
    adapter.Fill(emps);
}

private void Create(){
    DataRow r = emps.Tables["Table"].NewRow();
    r["FirstName"] = "Bob";
    r["LastName"] = "Dobbs";
    emps.Tables["Table"].Rows.Add(r);
    adapter.Update(emps);
}

private void Update(){
    DataRow aRow = emps.Tables["Table"].Rows[0];
    System.Console.WriteLine(
        "First Name: " + aRow["FirstName"]);
    string newName = null;
    if(aRow["FirstName"].Equals("Nancy")){
        newName = "Adam";
    }else{
        newName = "Nancy";
    }
    aRow.BeginEdit();
    aRow["FirstName"] = newName;
```

```
        aRow.EndEdit();
        System.Console.WriteLine(
            "First Name: " + aRow["FirstName"]);
        //Update only happens now
        int iChangedRows = adapter.Update(emps);
        System.Console.WriteLine("{0} rows updated",
                                 iChangedRows);
    }

    private void Reread(){
        adapter.Fill(emps);
    }

    private void Delete(){
        //Seems to return 1 greater than actual count
        int iRow = emps.Tables["Table"].Rows.Count;
        DataRow lastRow =
            emps.Tables["Table"].Rows[iRow - 1];
        System.Console.WriteLine("Deleting: " +
lastRow["FirstName"]);
        lastRow.Delete();
        int iChangedRows = adapter.Update(emps);
        System.Console.WriteLine("{0} rows updated",
            iChangedRows);
    }
}///:~
```

The **Main()** method outlines what we're going to do: read the "Employees" table, create a new record, update a record, reread the table (you can comment out the call to **Reread()** if you want to see a **DBConcurrencyException**), and delete the record we created.

The **Crud** class has instance variables for holding the **OleDbDataAdapter** and **DataSet** that the various methods will use. **ReadEmployees()** opens the database connection and creates the adapter just as we've done before.

The next line:

```
new OleDbCommandBuilder(adapter);
```

demonstrates a utility class that automatically generates and sets within the **OleDbDataAdapter** the SQL statements that insert, update, and

delete data in the same table acted on by the select command. **OleDbCommandBuilder** is very convenient for SQL data adapters that work on a single table (there's a corresponding **SqlCommandBuilder** for use with SQL Server). For more complex adapters that involve multiple tables, you have to set the corresponding **InsertCommand**, **DeleteCommand**, and **UpdateCommand** properties of the **OleDbDataAdapter**. These commands are needed to commit to the database changes made in the **DataSet.**

The first four lines of method **Create()** show operations on the **DataSet emps** that we've seen before – the use of **Table.NewRow(),** and **DataRowCollection.Add()** to manipulate the **DataSet**. The final line calls **IDataAdapter.Update()**, which attempts to commit the changes in the **DataSet** to the backing store (it is this method which requires the SQL commands generated by the **OleDbCommandBuilder**).

The method **Update()** begins by reading the first row in the **emps DataSet**. The call to **DataRow.BeginEdit()** puts the **DataRow** in a "Proposed" state. Changes proposed in a **DataRow** can either be accepted by a call to **DataRow.EndEdit()** or the **AcceptChanges()** method of either the **DataRow**, **DataTable**, or **DataSet**. They can be cancelled by a call to **DataRow.CancelEdit()** or the **RejectChanges()** methods of the classes.

After printing the value of the first row's "FirstName" column, we put **aRow** in a "Proposed" state and change the "FirstName" to "Fred." We call **CancelEdit()** and show on the console that "Fred" *is not* the value. If the first name is currently "Nancy" we're going to change it to "Adam" and vice versa. This time, after calling **BeginEdit()** and making the change, we call **EndEdit()**. At this point, the data is changed in the **DataSet**, but not yet in the database. The database commit is performed in the next line, with another call to **adapter.Update()**.

This call to **Update()** succeeds, as the rows operated on by the two calls to **Update()** are different. If, however, we were to attempt to update either of these two rows without rereading the data from the database, we would get the dread **DBConcurrencyException**. Since deleting the row we added is exactly our intent, **Main()** calls **Reread()** which in turn calls **adapter.Fill()** to refill the **emps DataSet**.

Finally, **Main()** calls **Delete()**. The number of rows is retrieved from the **Rows** collection. But because the index into rows is 0-based, we need to subtract 1 from the total count to get the index of the last row (e.g., the **DataRow** in a **DataTable** with a **Count** of 1 would be accessed at **Rows[0]**). Once we have the last row in the DataSet (which will be the "Bob Dobbs" record added by the **Create()** method), a call to **DataRow.Delete()** removes it from the **DataSet** and **DataAdapter.Update()** commits it to the database.

The Object-Relational Impedance Mismatch

If you ever find yourself unwelcome in a crowd of suspicious programmers, say "I was wondering what is your favorite technique for overcoming the object-relational impedance mismatch?" This is like a secret handshake in programmer circles: not only does it announce that you're not just some LISP hacker fresh from Kendall Square, it gives your inquisitors a chance to hold forth on A Matter of Great Import.

You can see the roots of the mismatch even in the basic examples we've shown here. It's taken us several pages just to show how to do the equivalent of **new** and assignment to relational data! Although a table is something like a class, and a row is something like an instance of the class, tables have no concept of binding data and behavior into a coherent whole, nor does the standard relational model have any concept of inheritance. Worse, it's become apparent over the years that there's no single strategy for mapping between objects and tables that is appropriate for all needs.

Thinking in Databases would be a very different book than *Thinking in C#*. The object and relational models are very different, but contain just enough similarities so that the pain hasn't been enough to trigger a wholesale movement towards object databases (which have been the Next Big Thing in Programming for more than a decade).

High-performing, highly-reliable object databases are available today, but have no mindshare in the enterprise market. What has gained mindshare is a hybrid model, which combines the repetitive structure of tables and rows with a hierarchical containment model that is closer to the object

model. This hybrid model, embodied in XML, does not directly support the more complicated concepts of relational joins or object inheritance, but is a good waypoint on the road to object databases. We'll discuss XML in more detail in Chapter #XML# and revisit ADO.NET in our discussion of data-bound controls in Chapter #Windows#.

Summary

To review the tools in the .NET Framework that collect objects:

1. An array associates numerical indices to objects. It holds objects of a known type so that you don't have to cast the result when you're looking up an object. It can be multidimensional in two ways – rectangular or jagged. However, its size cannot be changed once you create it.

2. An **IList** holds single elements, an **IDictionary** holds key-value pairs, and a **NameObjectCollectionBase** holds string-Collection pairs.

3. Like an array, an **IList** also associates numerical indices to objects—you can think of arrays and **IList**s as ordered containers. An **IDictionary** overloads the bracket operator of the array to make it easy to access values, but the underlying implementation is not necessarily ordered.

4. Most collections automatically resize themselves as you add elements, but the **BitArray** needs to be explicitly sized.

5. **ICollection**s hold only **object** references, so primitives are boxed and unboxed when stored. With the exception of type-specific containers in System.Collections.Specialized and those you roll yourself, you must always cast the result when you pull an **object** reference out of a container. Type-specific container classes will be supported natively by the .NET run-time sometime in the future.

6. Data structures have inherent characteristics distinct from the data that is stored in them. Sorting, searching, and traversal have traditionally been matters of great day-to-day import. Advances in abstraction and computer power allow most programmers to ignore

most of these issues most of the time, but occasionally produce the most challenging and rewarding opportunities in programming.

7. ADO.NET provides an abstraction of the relational database model. **DataSet**s represent relational data in memory, while **IDataAdapter**s and related classes move the data in and out of databases.

The collection classes are tools that you use on a day-to-day basis to make your programs simpler, more powerful, and more effective. Thoroughly understanding them and extending and combining them to rapidly solve solutions is one mark of software professionalism.

Exercises

10: Error Handling With Exceptions

In the previous chapter, the user began exploiting powerful, complex libraries to rapidly produce systems. Unfortunately, stuff happens. The concept of "robustness" is introduced and the reader learns that what separates good software from bad is not how well it does the job when things are going as expected, but how well it performs when the unthinkable happens. C#'s language features and the .NET framework's library features for dealing with exceptions are covered in depth.

This chapter ties C# together with "Extreme Programming," the set of software engineering disciplines that have gained widespread attention throughout the development world. The critical concept of "test-first coding" will be introduced, as well as tools for automating XP-style builds in C#.

Every program is based on a vast array of expectations. Some expectations are so basic that it doesn't make sense to worry about them – does the current computer have a hard-drive, sufficient RAM to load the program, and so forth. Other expectations are explicit in the code and violations can be discovered at compile-time – this method takes an integer as a parameter, not a string, that method returns a Fish not a Fowl. The majority of expectations, though, are implicit contracts between methods and the client code that calls them. When the reality at runtime is such that an expectation goes unfulfilled, C# uses Exceptions to signal the disruption of the program's expected behavior.

When an object can recognize a problem but does not have the context to intelligently deal with the problem, recovery may be possible. For instance, when a network message is not acknowledged, perhaps a retry is in order, but that decision shouldn't be made at the lowest level (network games, for instance, often have data of varying importance, some of which must be acknowledged and some which would be worthless by the time a retry could be made). On the other hand, a method may have a problem because something is awry with the way it is being used – perhaps a passed in parameter has an invalid value (a **PrintCalendar** method is called for the month "Eleventember") or perhaps the method can only be meaningfully called when the object is in a different state (for instance, a **Cancel** method is called when an **Itinerary** object is not in a "booked" state).

These misuse situations are tricky because there is no way in C# to specify a method's preconditions and postconditions as an explicit contract – a way in source code to say "if you call me with x, y, and z satisfied, I will guarantee that when I return condition a, b, and c will be satisfied (assuming of course that all the methods I call fulfill their contractual obligations with me)." For instance, .NET's Math class has a square root function that takes a double as its parameter. Since .NET does not have a class to represent imaginary numbers, this function can only return a meaningful answer if passed a positive value. If this method is called with

a negative value, is that an exceptional condition or a disappointing, but predictable, situation? There's no way to tell from the method's signature:

```
double Math.Sqrt(double d);
```

Although preconditions and postconditions are not explicit in C# code, you should always think in terms of contracts while programming and document pre- and post-conditions in your method's param and returns XML documentation. The .NET library writers followed this advice and the documentation for Math.Sqrt() explain that it will return a NaN (Not A Number) value if passed a negative parameter.

There is no hard-and-fast rule to determine what is an exceptional condition and what is reasonably foreseeable. Returning a special "invalid value" such as does Math.Sqrt() is debatable, especially if the precondition is not as obvious as "square roots can only be taken on positive numbers."

When an exceptional condition occurs such that a method cannot fulfill its post-conditions, there are only two valid things to do: attempt to change the conditions that led to the problem and retry the method, or "organized panic" – put objects into consistent states, close or release non-memory resources, and move control to a much different context that can either perform a recovery or log as much information as possible about the condition leading to the failure to help in debugging efforts. Some people emphasize recovery far too early; until late in the development of a high-availability system its better to have your system break and trigger a defect-fixing coding session than to cleanup-and-recover and allow the defect to continue.

Both of these valid choices (retrying or cleanup) usually cannot be fully done at the point where the exceptional condition occurred. With a network error sometimes just waiting a half-second or so and retrying may be appropriate, but usually a retry requires changing options at a higher level of abstraction (for instance, a file-writing related error might be retried after giving the user a chance to choose a different location). Similarly, cleanup leading to either recovery or an orderly shutdown may very well require behavior from all the objects in your system, not just those objects referenced by the class experiencing the problem.

When an exceptional condition occurs, it is up to the troubled method to create an object of a type derived from Exception. Such objects can be **thrown** so that control moves, not to the next line of code or into a method call as is normally the case, but rather propagates to blocks of code that are dedicated to the tasks of either recovery or cleanup.

The orderly way in which Exceptions propagate from the point of trouble has two benefits. First, it makes error-handling code hierarchical, like a chain of command. Perhaps one level of code can go through a sequence of options and retry, but if those fail, can give up and propagate the code to a higher level of abstraction, which may perform a clean shutdown. Second, exceptions clean up error handling code. Instead of checking for a particular rare failure and dealing with it at multiple places in your program, you no longer need to check at the point of the method call (since the exception will propagate right out of the problem area to a block dedicated to catching it). And, you need to handle the problem in only one place, the so-called *exception handler*. This saves you code, and it separates the code that describes what you want to do from the code that is executed when things go awry. In general, reading, writing, and debugging code becomes much clearer with exceptions than with alternative ways of error handling.

This chapter introduces you to the code you need to write to properly handle exceptions, and the way you can generate your own exceptions if one of your methods gets into trouble.

Basic exceptions

When you throw an exception, several things happen. First, the exception object is created in the same way that any C# object is created: on the heap, with **new**. Then the current path of execution (the one you couldn't continue) is stopped and the reference for the exception object is ejected from the current context. At this point the exception handling mechanism takes over and begins to look for an appropriate place to continue executing the program. This appropriate place is the *exception handler,* whose job is to recover from the problem so the program can either retry the task or cleanup and propagate either the original Exception or, better, a higher-abstraction Exception.

As a simple example of throwing an exception, consider an object reference called **t** that is passed in as a parameter to your method. Your design contract might require as a precondition that **t** refer to a valid, initialized object. Since C# has no syntax for enforcing preconditions, some other piece of code may pass your method a **null** reference and compile with no problem. This is an easy precondition violation to discover and there's no special information about the problem that you think would be helpful for its handlers. You can send information about the error into a larger context by creating an object representing the problem and its context and "throwing" it out of your current context. This is called *throwing an exception*. Here's what it looks like:

```
if(t == null)
  throw new ArgumentNullException();
```

This throws the exception, which allows you—in the current context—to abdicate responsibility for thinking about the issue further. It's just magically handled somewhere else. Precisely *where* will be shown shortly.

Exception arguments

Like any object in C#, you always create exceptions on the heap using **new**, which allocates storage and calls a constructor. There are four constructors in all standard exceptions:

- The default, no argument constructor
- A constructor that takes a string as a message:
  `throw new ArgumentNullException("t");`
- A constructor that takes a message and an inner, lower-level Exception:
  `throw new PreconditionViolationException("invalid t", new ArgumentNullException("t"))`
- And a constructor that allows for a lot of flexibility @todo forward reference to section dealing with SerializationInfo, StreamingContext constructor

The keyword **throw** causes a number of relatively magical things to happen. Typically, you'll first use **new** to create an object that represents

the error condition. You give the resulting reference to **throw**. The object is, in effect, "returned" from the method, even though that object type isn't normally what the method is designed to return. A simplistic way to think about exception handling is as an alternate return mechanism, although you get into trouble if you take that analogy too far. You can also exit from ordinary scopes by throwing an exception. But a value is returned, and the method or scope exits.

Any similarity to an ordinary return from a method ends here, because *where* you return is someplace completely different from where you return for a normal method call. (You end up in an appropriate exception handler that might be miles away—many levels lower on the call stack—from where the exception was thrown.)

Typically, you'll throw a different class of exception for each different type of error. The information about the error is represented both inside the exception object and implicitly in the type of exception object chosen, so someone in the bigger context can figure out what to do with your exception. (Often, it's fine that the only information is the type of exception object, and nothing meaningful is stored within the exception object.)

Catching an exception

If a method throws an exception, it must assume that exception is "caught" and dealt with. One of the advantages of C#'s exception handling is that it allows you to concentrate on the problem you're trying to solve in one place, and then deal with the errors from that code in another place.

To see how an exception is caught, you must first understand the concept of a *guarded region,* which is a section of code that might produce exceptions, and which is followed by the code to handle those exceptions.

The **try** block

If you're inside a method and you throw an exception (or another method you call within this method throws an exception), that method will exit in

the process of throwing. If you don't want a **throw** to exit the method, you can set up a special block within that method to capture the exception. This is called the *try block* because you "try" your various method calls there. The try block is an ordinary scope, preceded by the keyword **try**:

```
try {
  // Code that might generate exceptions
}
```

If you were checking for errors carefully in a programming language that didn't support exception handling, you'd have to surround every method call with setup and error testing code, even if you call the same method several times. With exception handling, you put everything in a try block and capture all the exceptions in one place. This means your code is a lot easier to write and easier to read because the goal of the code is not confused with the error checking.

Exception handlers

Of course, the thrown exception must end up someplace. This "place" is the *exception handler,* and there's one for every exception type you want to catch. Exception handlers immediately follow the try block and are denoted by the keyword **catch**:

```
try {
  // Code that might generate exceptions
} catch(Type1 id1) {
  // Handle exceptions of Type1
} catch(Type2 id2) {
  // Handle exceptions of Type2
} catch(Type3 id3) {
  // Handle exceptions of Type3
}

// etc...
```

Each catch clause (exception handler) is like a little method that takes one and only one argument of a particular type. The identifier (**id1**, **id2**, and so on) can be used inside the handler, just like a method argument. Sometimes you never use the identifier because the type of the exception

gives you enough information to deal with the exception, but the identifier must still be there.

The handlers must appear directly after the try block. If an exception is thrown, the exception handling mechanism goes hunting for the first handler with an argument that matches the type of the exception. Then it enters that catch clause, and the exception is considered handled. The search for handlers stops once the catch clause is finished. Only the matching catch clause executes; it's not like a **switch** statement in which you need a **break** after each **case**.

Note that, within the try block, a number of different method calls might generate the same exception, but you need only one handler.

@todo: Make sure that supertype matching is addressed near here

Exceptions have a helplink

Creating your own exceptions

You're not stuck using the existing C# exceptions. This is important because you'll often need to create your own exceptions to denote a special error that your library is capable of creating, but which was not foreseen when the C# exception hierarchy was created. C#'s predefined exceptions derive from SystemException, while your exceptions are expected to derive from ApplicationException.

To create your own exception class, you're forced to inherit from an existing type of exception, preferably one that is close in meaning to your new exception (this is often not possible, however). The most trivial way to create a new type of exception is just to let the compiler create the default constructor for you, so it requires almost no code at all:

```
//:c09:SimpleExceptionDemo.cs
// Inheriting your own exceptions.
using System;
```

```
class SimpleException : ApplicationException {}

public class SimpleExceptionDemo {
  public void F() {
    System.Console.WriteLine(
      "Throwing SimpleException from F()");
    throw new SimpleException ();
  }
  public static void Main() {
    SimpleExceptionDemo sed =
      new SimpleExceptionDemo();
    try {
      sed.F();
    } catch(SimpleException ) {
      System.Console.Error.WriteLine("Caught it!");
    }
  }
} ///:~
```

When the compiler creates the default constructor, it which automatically (and invisibly) calls the base-class default constructor. As you'll see, the most important thing about an exception is the class name, so most of the time an exception like the one shown above is satisfactory.

Here, the result is printed to the console *standard error* stream by writing to **System.Console.Error**. This stream can be redirected to any other **TextWriter** by calling **System.Console.SetError()** (note that this is 'asymmetric" – the **Error** property doesn't support assignment, but there's a **SetError()**. Why would this be?)

Creating an exception class that overrides the standard constructors is also quite simple:

```
//:c09:FullConstructors.cs
using System;

class MyException : Exception {
  public MyException() : base() {}
  public MyException(string msg) : base(msg) {}
  public MyException(string msg, Exception inner) :
base(msg, inner){}
```

```
}

public class FullConstructors {
  public static void F() {
    System.Console.WriteLine(
      "Throwing MyException from F()");
    throw new MyException();
  }
  public static void G()  {
    System.Console.WriteLine(
      "Throwing MyException from G()");
    throw new MyException("Originated in G()");
  }

  public static void H(){
      try{
        I();
      }catch(DivideByZeroException e){
        System.Console.WriteLine("Increasing
abstraction level");
        throw new MyException("Originated in H()", e);
      }
  }

  public static void I(){
      System.Console.WriteLine("This'll cause
trouble");
      int y = 0;
      int x = 1 / y;
  }

  public static void Main() {
    try {
      F();
    } catch(MyException e) {
        System.Console.Error.WriteLine(e.StackTrace);
    }
    try {
      G();
    } catch(MyException e) {
      System.Console.Error.WriteLine(e.Message);
```

```
    }
    try{
        H();
    }catch(MyException e){
        System.Console.Error.WriteLine(e.Message);
        System.Console.Error.WriteLine("Inner
exception: " + e.InnerException);
        System.Console.Error.WriteLine("Source: " +
e.Source);
        System.Console.Error.WriteLine("TargetSite: "
+ e.TargetSite);
    }
  }
} ///:~
```

The code added to **MyException** is small—the addition of three constructors that define the way **MyException** is created. The base-class constructor is explicitly invoked by using the **: base** keyword.

The output of the program is:

```
Throwing MyException from F()
   at FullConstructors.F()
   at FullConstructors.Main()
Throwing MyException from G()
Originated in G()
This'll cause trouble
Increasing abstraction level
Originated in H()
Inner exception: System.DivideByZeroException:
Attempted to divide by zero.
   at FullConstructors.I()
   at FullConstructors.H()
Source: FullConstructors
TargetSite: Void H()
```

You can see the absence of the detail message in the **MyException** thrown from **F( )**. The block that catches the exception thrown from **F()** shows the stack trace all the way to the origin of the exception. This is probably the most helpful property in Exception and is a great aid to debugging.

When **H()** executes, it calls **I()**, which attempts an illegal arithmetic operation. The attempt to divide by zero throws a **DivideByZeroException** (demonstrating the truth of the previous statement about the type name being the most important thing). **H()** catches the **DivideByZeroException**, but increases the abstraction level by wrapping it in a **MyException**. Then, when the **MyException** is caught in **Main()**, we can see the inner exception and its origin in **I()**.

The **Source** property contains the name of the assembly that threw the exception, while the **TargetSite** property returns a handle to the method that threw the exception. **TargetSite** is appropriate for sophisticated reflection-based exception diagnostics and handling.

The process of creating your own exceptions can be taken further. You can add extra constructors and members:

```
//:c09:ExtraFeatures.cs
// Further embellishment of exception classes.
using System;

class MyException2 : Exception {
  int errorCode;
  public int ErrorCode{
      get { return errorCode; }
  }

  public MyException2() : base(){}

  public MyException2(string msg) : base(msg) {}

  public MyException2(string msg, int errorCode) :
base(msg) {
    this.errorCode = errorCode;
  }
}

public class ExtraFeatures {
  public static void F() {
    System.Console.WriteLine(
      "Throwing MyException2 from F()");
```

```
    throw new MyException2();
  }
  public static void G()  {
    System.Console.WriteLine(
      "Throwing MyException2 from G()");
    throw new MyException2("Originated in G()");
  }
  public static void H() {
    System.Console.WriteLine(
      "Throwing MyException2 from H()");
    throw new MyException2(
      "Originated in H()", 47);
  }
  public static void Main(String[] args) {
    try {
      F();
    } catch(MyException2 e) {
        System.Console.Error.WriteLine(e.StackTrace);
    }
    try {
      G();
    } catch(MyException2 e) {
        System.Console.Error.WriteLine(e.StackTrace);
    }
    try {
      H();
    } catch(MyException2 e) {
      System.Console.Error.WriteLine(e.StackTrace);
      System.Console.Error.WriteLine("e.ErrorCode = "
+ e.ErrorCode);
    }
  }
} ///:~
```

A property **ErrorCode** has been added, along with an additional constructor that sets it. The output is:

```
Throwing MyException2 from F()
   at ExtraFeatures.F() in
D:\tic\exceptions\ExtraFeatures.cs:line 23
```

```
   at ExtraFeatures.Main(String[] args) in
C:\Documents and Settings\larry\My
Documents\ExtraFeatures.cs:line 38
Throwing MyException2 from G()
   at ExtraFeatures.G() in
D:\tic\exceptions\ExtraFeatures.cs:line 28
   at ExtraFeatures.Main(String[] args) in
D:\tic\exceptions\ExtraFeatures.cs:line 43
Throwing MyException2 from H()
   at ExtraFeatures.H() in
D:\tic\exceptions\ExtraFeatures.cs:line 33
   at ExtraFeatures.Main(String[] args) in
C:\Documents and Settings\larry\My
Documents\ExtraFeatures.cs:line 48
e.ErrorCode = 47
```

Since an exception is just another kind of object, you can continue this process of embellishing the power of your exception classes. An error code, as illustrated here, is probably not very useful, since the type of the Exception gives you a good idea of the "what" of the problem. More interesting would be to embed a clue as to the "how" of retrying or cleanup – some kind of object that encapsulated the context of the broken contract. Keep in mind, however, that the best design is the one that throws exceptions rarely and that the best programmer is the one whose work delivers the most benefit to the customer, not the one who comes up with the cleverest solution to what to do when things go wrong!

C#'s Lack Of Checked Exceptions

Some languages, notably Java, require a method to list recoverable exceptions it may throw. Thus, in Java, reading data from a stream is done with a method that is declared as `int read() throws IOException` while the equivalent method in C# is simply `int read()`. This does not mean that C# somehow avoids the various unforeseeable circumstances that can ruin a read, nor even that they are necessarily less

likely to occur in C#. If you look at the documentation for `System.IO.Stream.Read()` you'll see that it can throw, yes, an **IOException**.

The effect of including a list of exceptions in a method's declaration is that at compile-time, if the method is used, the compiler can assure that the Exception is either handled or passed on. Thus, in languages like Java, the exception is explicitly part of the method's signature – "Pass me parameters of type such and so and I'll return a value of a certain type. However, if things go awry, I may also throw these particular types of Exception." Just as the compiler can enforce that a method that takes an int and returns a string is not passed a double or used to assign to a float so too does the compiler enforce the Exception specification.

Checked exceptions such as in Java are not intended to deal with precondition violations, which are by far the most common cause of exceptional conditions. A precondition violation (calling a method with an improper parameter, calling a state-specific method on an object that's not in the required state) is, by definition, the result of a programming error. Retries are, at best, useless in such a situation (at worst, the retry will work and thereby allow the programming error to go unfixed!). So Java has another type of exception that is unchecked.

In practice what happens is that while programmers are generally accepting of strong type-checking when it comes to parameters and return values, the value of strongly typed exceptions is not nearly as evident in real-world practice. There are too many low-level things that can go wrong (failures of files and networks and RAM and so forth) and many programmers do not see the benefit of creating an abstraction hierarchy as they deal with all failures in a generic manner. And the different intent of checked and unchecked exceptions is confusing to many developers.

And thus one sees a great deal of Java code in two equally bad forms: meaningless propagation of low-abstraction exceptions (Web services that are declared as throwing IOExceptions) and "make it compile" hacks where methods are declared as "throws Exception" (in other words, saying "I can throw anything I darn well please."). Worse, though, it's not uncommon to see the very worst possible "solution," which is to catch and ignore the exception, all for the sake of getting a clean compile.

Theoretically, if you're going to have a strongly typed language, it probably makes sense for exceptions to be part of the method signature. Pragmatically, though, the prevalence of bad exception-handling code in Java argues for C#'s approach, which is essentially that the burden is on the programmer to know to place error-handling code in the appropriate places.

Catching any exception

It is possible to create a handler that catches any type of exception. You do this by catching the base-class exception type **Exception**:

```
catch(Exception e) {
  System.Console.Error.WriteLine("Caught an
exception");
}
```

This will catch any exception, so if you use it you'll want to put it at the *end* of your list of handlers to avoid preempting any exception handlers that might otherwise follow it.

Since the **Exception** class is the base of all the exception classes, you don't get much specific information about the specific problem. You do, however, get some methods from **object** (everybody's base type). The one that might come in handy for exceptions is **GetType()**, which returns an object representing the class of **this**. You can in turn read the **Name** property of this **Type** object. You can also do more sophisticated things with **Type** objects that aren't necessary in exception handling. **Type** objects will be studied later in this book.

Rethrowing an exception

Sometimes you'll want to rethrow the exception that you just caught, particularly when you use **catch(Exception)** to catch any exception. Since you already have the reference to the current exception, you can simply rethrow that reference:

```
catch(Exception e) {
  System.Console.Error.WriteLine("An exception was
thrown");
  throw e;
```

```
}
```

Rethrowing an exception causes the exception to go to the exception handlers in the next-higher context. Any further **catch** clauses for the same **try** block are still ignored. In addition, everything about the exception object is preserved, so the handler at the higher context that catches the specific exception type can extract all the information from that object.

Elevating the abstraction level

Usually when catching exceptions and then propagating them outward, you should elevate the abstraction level of the caught Exception. For instance, at the business-logic level, all you may care about is that "the charge didn't go through." You'll certainly want to preserve the information of the less-abstract Exception for debugging purposes, but for logical purposes, you want to deal with all problems equally.

```
//:c09:Rethrow.cs
using System;

namespace Rethrow{
    class TransactionFailureException :
ApplicationException {
        public TransactionFailureException(Exception
e) :
        base("Logical failure caused by low-level
exception", e){
        }
    }

    class Transaction {
        Random r = new Random();
        public void Process(){
            try {
                if (r.NextDouble() > 0.3) {
                    throw new ArithmeticException();
                } else {
                    if (r.NextDouble() > 0.5) {
                        throw new FormatException();
                    }
```

```
                }

            } catch (Exception e) {
                TransactionFailureException tfe = new
TransactionFailureException(e);
                throw tfe;
            }
        }
    }


    class BusinessLogic {
        Transaction myTransaction = new Transaction();

        public void DoCharge(){
            try {
                myTransaction.Process();
                System.Console.WriteLine("Transaction
went through");
            } catch (TransactionFailureException tfe)
{
                System.Console.WriteLine(tfe.Message);

System.Console.Error.WriteLine(tfe.InnerException);
            }
        }

        public static void Main(){
            BusinessLogic bl = new BusinessLogic();
            for (int i = 0; i < 10; i++) {
                bl.DoCharge();
            }
        }
    }
} ///:~
```

In this example, the class **Transaction** has an exception class that is at its same level of abstraction in **TransactionFailureException**. The **try...catch(Exception e**) construct in **Transaction.Process()** makes for a nice and explicit contract: "I try to return void, but if anything goes

awry in my processing, I may throw a **TransactionFailedException**." In order to generate some exceptions, we use a random number generator to throw different types of low-level exceptions in **Transaction.Process()**.

All exceptions are caught in **Transaction.Process()**'s catch block, where they are placed "inside" a new **TransactionFailureException** using that type's overridden constructor that takes an exception and creates a generic "Logical failure caused by low-level exception" message. The code then throws the newly created **TransactionFailureException**, which is in turn caught by **BusinessLogic.DoCharge()**'s **catch(TransactionFailureException tfe)** block. The higher-abstraction exception's message is printed to the Console, while the lower-abstraction exception is sent to the Error stream (which is also the console, but the point is that there is a separation between the two levels of abstraction. In practice, the higher-abstraction exception would be used for business logic choices and the lower-abstraction exception for debugging).

Standard C# exceptions

The C# class **Exception** describes anything that can be thrown as an exception. There are two general types of **Exception** objects ("types of" = "inherited from"). **SystemException** represents exceptions in the System namespace and its descendants (in other words, .NET's standard exceptions). Your exceptions by convention should extend from **ApplicationException**.

If you browse the .NET documentation, you'll see that each namespace has a small handful of exceptions that are at a level of abstraction appropriate to the namespace. For instance, System.IO has an **InternalBufferOverflowException**, which is pretty darn low-level, while System.Web.Services.Protocols has **SoapException**, which is pretty darn high-level. It's worth browsing through these exceptions once to get a feel for the various exceptions, but you'll soon see that there isn't anything special between one exception and the next except for the name.

The basic idea is that the name of the exception represents the problem that occurred, and the exception name is intended to be relatively self-explanatory.

Performing cleanup with finally

There's often some piece of code that you want to execute whether or not an exception is thrown within a **try** block. This usually pertains to some operation other than memory recovery (since that's taken care of by the garbage collector). To achieve this effect, you use a **finally** clause at the end of all the exception handlers. The full picture of an exception handling section is thus:

```
try {
  // The guarded region: Dangerous activities
  // that might throw A, B, or C
} catch(A a1) {
  // Handler for situation A
} catch(B b1) {
  // Handler for situation B
} catch(C c1) {
  // Handler for situation C
} finally {
  // Activities that happen every time
}
```

In **finally** blocks, you can use control flow statements **break, continue,** or **goto** only for loops that are entirely inside the **finally** block; you cannot perform a jump out of the **finally** block. Similarly, you can not use **return** in a **finally** block. Violating these rules will give a compiler error.

To demonstrate that the **finally** clause always runs, try this program:

```
//:c09:AlwaysFinally.cs
// The finally clause is always executed.
using System;
```

```
class ThreeException : ApplicationException {}

public class FinallyWorks {
  static int count = 0;
  public static void Main() {
      while(true){
          try{
             if(count++ < 3){
                throw new ThreeException();
             }
            System.Console.WriteLine("No exception");
      } catch(ThreeException ) {
        System.Console.WriteLine("ThreeException");
      } finally {
        System.Console.Error.WriteLine("In finally
clause");
        //! if(count == 3) break; <- Compiler error
      }
      if(count > 3)
          break;
    }
  }
} ///:~
```

This program also gives a hint for how you can deal with the fact that exceptions in C# do not allow you to resume back to where the exception was thrown, as discussed earlier. If you place your **try** block in a loop, you can establish a condition that must be met before you continue the program. You can also add a **static** counter or some other device to allow the loop to try several different approaches before giving up. This way you can build a greater level of robustness into your programs.

The output is:

```
ThreeException
In finally clause
ThreeException
In finally clause
ThreeException
In finally clause
No exception
```

```
In finally clause
```

Whether an exception is thrown or not, the **finally** clause is always executed.

What's **finally** for?

Since C# has a garbage collector, releasing memory is virtually never a problem. So why do you need **finally**?

finally is necessary when you need to set something *other* than memory back to its original state. This is some kind of cleanup like an open file or network connection, something you've drawn on the screen, or even a switch in the outside world, as modeled in the following example:

```
//:c09:WhyFinally.cs
// Why use finally?
using System;

class Switch {
  bool state = false;
  public bool Read{
      get { return state; }
      set { state = value; }
  }

  public void On(){ state = true; }
  public void Off(){ state = false; }
}
class OnOffException1 : Exception {}
class OnOffException2 : Exception {}

public class OnOffSwitch {
  static Switch sw = new Switch();
  static void F() {}
  public static void Main() {
    try {
      sw.On();
      // Code that can throw exceptions...
      F();
      sw.Off();
    } catch(OnOffException1 ) {
```

```
      System.Console.WriteLine("OnOffException1");
      sw.Off();
    } catch(OnOffException2 ) {
      System.Console.WriteLine("OnOffException2");
      sw.Off();
    }
  }
} ///:~
```

The goal here is to make sure that the switch is off when **Main( )** is completed, so **sw.Off( )** is placed at the end of the try block and at the end of each exception handler. But it's possible that an exception could be thrown that isn't caught here, so **sw.Off( )** would be missed. However, with **finally** you can place the cleanup code from a try block in just one place:

```
//:c09:WhyFinally2.cs
// Why use finally?
using System;

class Switch {
    bool state = false;
    public bool Read{
        get { return state;}
        set { state = value;}
    }

    public void On(){ state = true;}
    public void Off(){ state = false;}
}
class OnOffException1 : Exception {
}
class OnOffException2 : Exception {
}

public class OnOffSwitch {
    static Switch sw = new Switch();
    static void F() {}
    public static void Main() {
        try {
            sw.On();
```

```
            // Code that can throw exceptions...
            F();
        } catch (OnOffException1 ) {

System.Console.WriteLine("OnOffException1");
        } catch (OnOffException2 ) {

System.Console.WriteLine("OnOffException2");
        } finally {
            sw.Off();
        }
    }
} ///:~
```

Here the **sw.Off( )** has been moved to just one place, where it's guaranteed to run no matter what happens.

Even in cases in which the exception is not caught in the current set of **catch** clauses, **finally** will be executed before the exception handling mechanism continues its search for a handler at the next higher level:

```
//:c09:NestedFinally.cs
// Finally is always executed.
using System;

class FourException : ApplicationException {}

public class AlwaysFinally {
  public static void Main() {
    System.Console.WriteLine(
      "Entering first try block");
    try {
      System.Console.WriteLine(
        "Entering second try block");
      try {
        throw new FourException();
      } finally {
        System.Console.WriteLine(
          "finally in 2nd try block");
      }
    } catch(FourException ) {
```

```
      System.Console.WriteLine(
        "Caught FourException in 1st try block");
    } finally {
      System.Console.WriteLine(
        "finally in 1st try block");
    }
  }
} ///:~
```

The output for this program shows you what happens:

```
Entering first try block
Entering second try block
finally in 2nd try block
Caught FourException in 1st try block
finally in 1st try block
```

Finally and **using**

Way back in Chapter #initialization and cleanup#, we discussed C#'s **using** blocks. Now we can finally explain how it works. Consider this code, which uses inheritance, upcasting, and a **try...finally** block to ensure that cleanup happens:

```
//:c09:UsingCleanup.cs
using System;

class UsingCleanup : IDisposable {
    public static void Main(){
        try {
            IDisposable uc = new UsingCleanup();
            try {
                throw new NotImplementedException();
            } finally {
                uc.Dispose();
            }
        }catch(Exception ){
            System.Console.WriteLine("After
disposal");
        }
    }
```

```
    UsingCleanup(){
        System.Console.WriteLine("Constructor
called");
    }

    public void Dispose(){
        System.Console.WriteLine("Dispose called");
    }

    ~UsingCleanup(){
        System.Console.WriteLine("Destructor called");
    }
}//:~
```

You should not be surprised at the output:

```
Constructor called
Dispose called
After disposal
Destructor called
```

Changing the **Main()** method to:

```
    public static void Main(){
        try {
            UsingCleanup uc = new UsingCleanup();
            using(uc){
                throw new NotImplementedException();
            }
        }catch(Exception ){
            System.Console.WriteLine("After
disposal");
        }
    }
```

produces the exact same output. In fact, the **using** keyword is just "syntactic sugar" that wraps an **IDisposable** subtype in a **try...finally** block! Behind the scenes, the exact same code is generated, but the **using** block is terser.

Pitfall: the lost exception

In general, C#'s exception implementation is quite outstanding, but unfortunately there's a flaw. Although exceptions are an indication of a crisis in your program and should never be ignored, it's possible for an exception to simply be lost. This happens with a particular configuration using a **finally** clause:

```
//:c09:LostException.cs
// How an exception can be lost.
using System;

class VeryImportantException : Exception {
}

class HoHumException : Exception {
}

public class LostMessage {
    void F() {
        throw new VeryImportantException();
    }
    void Dispose()  {
        throw new HoHumException();
    }
    public static void Main(){
        try {
            LostMessage lm = new LostMessage();
            try {
                lm.F();
            } finally {
                lm.Dispose();
            }
        } catch (Exception e) {
            System.Console.WriteLine(e);
        }
    }
} ///:~
```

The output is:

```
HoHumException: Exception of type HoHumException was
thrown.
   at LostMessage.Dispose()
   at LostMessage.Main()
```

You can see that there's no evidence of the **VeryImportantException**, which is simply replaced by the **HoHumException** in the **finally** clause. This is a rather serious pitfall, since it means that an exception can be completely lost, and in a far more subtle and difficult-to-detect fashion than the example above. In contrast, C++ treats the situation in which a second exception is thrown before the first one is handled as a dire programming error. To avoid this possibility, it is a good idea to wrap all your work inside a **finally** block in a **try...catch(Exception):**


```
//:c09:CarefulFinally.cs
using System;

class VeryImportantException : Exception {
}

class HoHumException : Exception {
}

public class LostMessage {
    void F() {
        throw new VeryImportantException();
    }
    void Dispose()  {
        throw new HoHumException();
    }
    public static void Main(){
        try {
            LostMessage lm = new LostMessage();
            try {
                lm.F();
            } finally {
                try{
                    lm.Dispose();
                }catch(Exception e){
                    System.Console.WriteLine(e);
                }
            }
```

```
        } catch (Exception e) {
            System.Console.WriteLine(e);
        }
    }
} ///:~
```

Produces the desired output:

```
HoHumException: Exception of type HoHumException was
thrown.
   at LostMessage.Dispose()
   at LostMessage.Main()
VeryImportantException: Exception of type
VeryImportantException was thrown.
   at LostMessage.F()
   at LostMessage.Main()
```

Constructors

When writing code with exceptions, it's particularly important that you always ask, "If an exception occurs, will this be properly cleaned up?" Most of the time you're fairly safe, but in constructors there's a problem. The constructor puts the object into a safe starting state, but it might perform some operation—such as opening a file—that doesn't get cleaned up until the user is finished with the object and calls a special cleanup method. If you throw an exception from inside a constructor, these cleanup behaviors might not occur properly. This means that you must be especially diligent while you write your constructor.

Since you've just learned about **finally**, you might think that it is the correct solution. But it's not quite that simple, because **finally** performs the cleanup code *every time,* even in the situations in which you don't want the cleanup code executed until the cleanup method runs. Thus, if you do perform cleanup in **finally**, you must set some kind of flag when the constructor finishes normally so that you don't do anything in the **finally** block if the flag is set. Because this isn't particularly elegant (you are coupling your code from one place to another), it's best if you try to avoid performing this kind of cleanup in **finally** unless you are forced to.

In the following example, a class called **InputFile** is created that opens a file and allows you to read it one line (converted into a **String**) at a time. It uses the classes **FileReader** and **BufferedReader** from the Java standard I/O library that will be discussed in Chapter 11, but which are simple enough that you probably won't have any trouble understanding their basic use:

```
//:c09:ConstructorFinally.cs
// Paying attention to exceptions
// in constructors.
using System;
using System.IO;

namespace Cleanup{
    internal class InputFile : IDisposable {
        private StreamReader inStream;
        internal InputFile(string fName) {
            try {
                inStream =
                new StreamReader(
                                new FileStream(
                                                fName,
FileMode.Open));
                // Other code that might throw
exceptions
            } catch (FileNotFoundException e) {
                System.Console.Error.WriteLine(
                                                "Could
not open " + fName);
                // Wasn't open, so don't close it
                throw e;
            } catch (Exception e) {
                // All other exceptions must close it
                try {
                    inStream.Close();
                } catch (IOException ) {
                    System.Console.Error.WriteLine(

"in.Close() unsuccessful");
                }
                throw e; // Rethrow
```

```
            } finally {
                // Don't close it here!!!
            }
        }
        internal string ReadLine() {
            string s;
            try {
                s = inStream.ReadLine();
            } catch (IOException ) {
                System.Console.Error.WriteLine(

"ReadLine() unsuccessful");
                s = "failed";
            }
            return s;
        }
        public void Dispose() {
            try {
                inStream.Close();
            } catch (IOException ) {
                System.Console.Error.WriteLine(

"in.Close() unsuccessful");
            }
        }
    }

    public class Cleanup {
        public static void Main() {
            try {
                InputFile inFile =
                new InputFile("Cleanup.cs");
                using(inFile){
                    String s;
                    int i = 1;
                    while ((s = inFile.ReadLine()) !=
null)
                        System.Console.WriteLine(
                                                ""+
i++ + ": " + s);
                }
```

```
            } catch (Exception e) {
                System.Console.Error.WriteLine(
                                                "Caught
in Main");
                System.Console.Error.WriteLine(

e.StackTrace);
            }
        }
    }
} ///:~
```

The constructor for **InputFile** takes a **string** argument, which is the name of the file you want to open. Inside a **try** block, it creates a **FileStream** using the file name. A **FileStream** isn't particularly useful for text until you turn around and use it to create a **StreamReader** that can deal with more than one character at a time.

If the **FileStream** constructor is unsuccessful, it throws a **FileNotFoundException**, which must be caught separately because that's the one case in which you don't want to close the file since it wasn't successfully opened. Any *other* catch clauses must close the file because it *was* opened by the time those catch clauses are entered. (Of course, this is trickier if more than one method can throw a **FileNotFoundException**. In that case, you might want to break things into several **try** blocks.) The **Close( )** method might throw an exception so it is tried and caught even though it's within the block of another **catch** clause—it's just another pair of curly braces to the C# compiler. After performing local operations, the exception is rethrown, which is appropriate because this constructor failed, and you wouldn't want the calling method to assume that the object had been properly created and was valid.

In this example, which doesn't use the aforementioned flagging technique, the **finally** clause is definitely *not* the place to **Close( )** the file, since that would close it every time the constructor completed. Since we want the file to be open for the useful lifetime of the **InputFile** object this would not be appropriate.

The **ReadLine( )** method returns a **string** containing the next line in the file. It calls **StreamReader.ReadLine( ),** which can throw an

exception, but that exception is caught so **ReadLine( )** doesn't throw any exceptions.

The **Dispose( )** method must be called when the **InputFile** is finished with. This will release the system resources (such as file handles) that are used by the **StreamReader** and/or **FileStream** objects. You don't want to do this until you're finished with the **InputFile** object, at the point you're going to let it go. You might think of putting such functionality into a destructor method, but as mentioned in Chapter #initialization and cleanup# you can't always be sure when the destructor will be called (even if you *can* be sure that it will be called, all you know about *when* is that it's sure to be called before the process ends).

In the **Cleanup** class, an **InputFile** is created to open the same source file that creates the program, the file is read in a line at a time, and line numbers are added. The **using** keyword is used to ensure that **InputFile.Dispose()** is called. All exceptions are caught generically in **Main( )**, although you could choose greater granularity.

One of the benefits of this example is to show you why exceptions are introduced at this point in the book—you can't do basic I/O without using exceptions. Exceptions are so integral to programming in C# that you can accomplish only so much without knowing how to work with them.

Exception matching

When an exception is thrown, the exception handling system looks through the "nearest" handlers in the order they are written. When it finds a match, the exception is considered handled, and no further searching occurs.

Matching an exception doesn't require a perfect match between the exception and its handler. A derived-class object will match a handler for the base class, as shown in this example:

```
//:c09:Sneeze.cs
// Catching exception hierarchies.
using System;

class Annoyance : Exception {}
```

```
class Sneeze : Annoyance {}

public class Human {
  public static void Main() {
    try {
      throw new Sneeze();
    } catch(Sneeze ) {
      System.Console.Error.WriteLine("Caught Sneeze");
    } catch(Annoyance ) {
      System.Console.Error.WriteLine("Caught
Annoyance");
    }
  }
} ///:~
```

The **Sneeze** exception will be caught by the first **catch** clause that it matches—which is the first one, of course. However, if you remove the first catch clause, leaving only:

```
    try {
      throw new Sneeze();
    } catch(Annoyance) {
      System.Console.Error.WriteLine("Caught
Annoyance");
    }
```

The code will still work because it's catching the base class of **Sneeze**. Put another way, **catch(Annoyance e)** will catch an **Annoyance** *or any class derived from it*. This is useful because if you decide to add more derived exceptions to a method, then the client programmer's code will not need changing as long as the client catches the base class exceptions.

If you try to "mask" the derived-class exceptions by putting the base-class catch clause first, like this:

```
    try {
      throw new Sneeze();
    } catch(Annoyance a) {
      System.Console.Error.WriteLine("Caught
Annoyance");
```

```
    } catch(Sneeze s) {
      System.Console.Error.WriteLine("Caught Sneeze");
    }
```

the compiler will give you an error message, since it sees that the **Sneeze** catch-clause can never be reached.

Exception guidelines

Use exceptions to:

8. Fix the problem and call the method that caused the exception again.
9. Patch things up and continue without retrying the method.
10. Calculate some alternative result instead of what the method was supposed to produce.
11. Do whatever you can in the current context and rethrow the *same* exception to a higher context.
12. Do whatever you can in the current context and throw a *different* exception to a higher context.
13. Terminate the program.
14. Simplify. (If your exception scheme makes things more complicated, then it is painful and annoying to use.)
15. Make your library and program safer. (This is a short-term investment for debugging, and a long-term investment (for application robustness.)

Summary

Improved error recovery is one of the most powerful ways that you can increase the robustness of your code. Error recovery is a fundamental concern for every program you write, but it's especially important in C#, where one of the primary goals is to create program components for others to use. *To create a robust system, each component must be robust.*

Exceptions are not terribly difficult to learn, and are one of those features that provide immediate and significant benefits to your project.

@todo – New Chapter? Design By Contract

Whereas C# has no checked exceptions, and Java has checked exceptions that enforce the propagation and handling of those exceptions based on bad program state, but does nothing to diagnose precondition and postcondition violations, the Eiffel programming allows one to specify a method's contract explicitly. The concept of an explicit declaration of a method's contract and a largely automatic way to validate that contract lies at the heart of "Extreme Programming," the set of disciplines which have recently gained great attention.

The essential *discipline* in Extreme Programming is called "test-first programming." The tests, in this case, are unit tests – exercises of a class's set of contracts. The acceptability of this discipline is dependent upon making the creation and execution of test suites as straightforward as possible; programmers will not write test code if doing so is not very easy.

Using C#'s attributes, it is possible for a framework to "discover" methods that are marked as tests.

Exercises

11: I/O in C#

Creating a good input/output (I/O) system is one of the more difficult tasks for the language designer.

This is evidenced by the number of different approaches. The challenge seems to be in covering all eventualities. Not only are there different sources and sinks of I/O that you want to communicate with (files, the console, network connections), but you need to talk to them in a wide variety of ways (sequential, random-access, buffered, binary, character, by lines, by words, etc.).

The .NET library designers attacked this problem by creating lots of classes. In fact, there are so many classes for .NET's I/O system that it can be intimidating at first (ironically, the design actually prevents an explosion of classes). As a result there are a fair number of classes to learn before you understand enough of .NET's I/O picture to use it properly.

File, Directory, and Path

Before getting into the classes that actually read and write data to streams, we'll look at the utility classes that assist you in handling file directory issues. These utility classes consist of three classes that have just static methods: **Path**, **Directory**, and **File**. These classes have somewhat confusing names in that there's no correspondence between object instances and items within the file-system (indeed, you can't instantiate these classes – their constructors are not public).

A directory lister

Let's say you want to list the names of the files in the directory. This is easily done with the static **Directory.GetFiles()** method, as is shown in this sample:

```
//c11:FList.cs
//Displays directory listing
```

```
using System.IO;

public class FList{
   public static void Main(string[] args){
      string dirToRead = ".";
      string pattern = "*";
      if(args.Length > 0){
         dirToRead = args[0];
      }
      if(args.Length > 1){
         pattern = args[1];
      }
      string[] fPaths =
         Directory.GetFiles(dirToRead, pattern);
      foreach(string fPath in fPaths){
         FileInfo fInfo = new FileInfo(fPath);
         System.Console.WriteLine(
            "Path = {0} Filename: {1} ext: {2} "
            + "touch: {3} size: {4}",
            fPath, fInfo.Name, fInfo.Extension,
            fInfo.LastWriteTime, fInfo.Length);
         System.Console.WriteLine(fName);
      }
   }
}///:~
```

When run with no arguments, this returns all the names in the current directory; another directory can be specified with the first argument, and a standard DOS filename pattern with the second. An overload of the **Directory.GetFiles()** method takes just a single string, which is equivalent to calling **Directory.GetFiles(dirString, "*").**

The method **Directory.GetFiles()** is a little poorly named, it's really returned strings that represent the paths to files (so perhaps it would have been better named **GetFilePaths()**). To get information on the corresponding file, the path is passed in to a **FileInfo** constructor. The **FileInfo** and related **DirectoryInfo** classes encapsulate what the file system knows – things like size, time of creation and last edit, attributes such as being ReadOnly, etc.

Checking for and creating directories

The **FList** program above returned only paths to files, not to directories. **Directory.GetFileSystemEntries()** returns paths to both files and directories, while **Directory.GetDirectories()** returns paths to the subdirectories of the given directory.

```
//c11: DirList.cs
//Displays listing of subdirectories
using System.IO;

public class DirList{
   public static void Main(string[] args){
      string dirToRead = ".";
      string pattern = "*";
      if(args.Length > 0){
         dirToRead = args[0];
      }
      if(args.Length > 1){
         pattern = args[1];
      }
      string[] subdirs =
         Directory.GetDirectories(
            dirToRead, pattern);
      foreach(string subdir in subdirs){
         DirectoryInfo dInfo =
            new DirectoryInfo(subdir);
         System.Console.WriteLine(
           "Path = {0} Created: {1} Accessed: {2} "
           + " Written to {3} ",
           subdir, dInfo.CreationTime,
           dInfo.LastAccessTime, dInfo.LastWriteTime);
      }
   }
}///:~
```

In addition to getting information on files and directories, the **File** and **Directory** classes contain methods for creating, deleting, and moving filesystem entities. This example shows how to create new subdirectories, files, and delete directories:

```
//:c11:FileManip
//Demonstrates basic filesystem manipulation
using System;
using System.IO;

class FileManip{
   public static void Main(){
      string curPath =
        Directory.GetCurrentDirectory();
      DirectoryInfo curDir =
         new DirectoryInfo(curPath);
      string curName = curDir.Name;
      char[] chars = curName.ToCharArray();
      Array.Reverse(chars);
      string revName = new String(chars);
      if(Directory.Exists(revName)){
         Directory.Delete(revName, true);
      }else{
         Directory.CreateDirectory(revName);
         string fName = "./" + revName + "/Foo.file";
         File.Create(fName);
      }
   }
}///:~
```

First, we use **Directory.GetCurrentDirectory()** to retrieve the current path; the same data is also available as **Environment.CurrentDirectory**. To get the current directory's name, we use the **DirectoryInfo** class and its **Name** property. The name of our new directory is the current directory's name in reverse. The first time this program runs, **Directory.Exists()** will return false (unless you happen to run this program in a directory with a reversed-name subdirectory already there!). In that case, we use **Directory.CreateDirectory()** and **File.Create()** (note the slight inconsistency in naming) to create a new subdirectory and a file. If you check, you'll see that "Foo.file" is of length 0 – **File.Create()** works at the filesystem level, not at the level of actually initializing the file with useful data.

The second time **FileManip** is run, the **Exists()** method will return true and **Directory.Delete()** deletes both the directory *and all its contents, including files and subdirectories*. If you don't want this highly

dangerous behavior, either pass in a **false** value, or use the overloaded **Directory.Delete(string)** which doesn't take a bool and which will throw an **IOException** if called on a non-empty directory.

Isolated Stores

Some of the most common file-oriented tasks are associated with a single user: preferences should be set to individual users, security dictates that there be restrictions on arbitrary file manipulation by components downloaded off the Web, etc. In these scenarios, the .NET Framework provides for *isolated storage*. An isolated store is a virtual file system within a *data compartment*. A data compartment is based on the user, assembly, and perhaps other aspects of the code's identity (e.g., it's signature). Isolated storage is for those situations when you don't need or want database-level security and control; isolated stores end up as files on the hard-drive and while operating-system restrictions may prevent them from being casually available, it's not appropriate to use isolated storage for high-value data.

Getting an isolated store for the current assembly is straightforward if wordy, one uses a static method called **GetUserStoreForAssembly()** in the **IsolatedStorageFile** class:

```
//c11:IsoStore.cs
using System.IO.IsolatedStorage;

class IsoStore{
   public static void Main(){
      IsolatedStorageFile isf =
        IsolatedStorageFile.GetUserStoreForAssembly();
      System.Console.WriteLine(
         "Assembly identity {0} \n" +
         "CurrentSize {1} \n" +
         "MaximumSize {2} \n" +
         "Scope {3} \n",
         isf.AssemblyIdentity,
         isf.CurrentSize, isf.MaximumSize, isf.Scope);

   }
}///:~
```

First, we have to specify that we're using the **System.IO.IsolatedStorage** namespace. After we create the isolated store, we print some of its attributes to the console. A typical run looks like this:

```
Assembly identity <System.Security.Policy.Url
version="1">
   <Url>file://D:/tic/chap11/IsoStore.exe</Url>
</System.Security.Policy.Url>

CurrentSize 0
MaximumSize 9223372036854775807
Scope User, Assembly
```

Because we've not done any digital signing (see chapter #security#), the *identity* of the assembly this is being run from is simply the name of the assembly as a URL. The store consumes no space currently and would be allowed to consume as much as 9GB. The store that we've got a handle on is associated with the user and assembly's identity; if we changed either of those, we'd get a different isolated store.

The **IsolatedFileStore** is essentially a virtual file system, but unfortunately it does not support the general **System.IO** classes such as **Directory**, **DirectoryInfo**, **File**, and **FileInfo**. Rather, the **IsolatedFileStore** class has static methods **GetDirectoryNames()** and **GetFileNames()** which correspond to **Directory.GetDirectories()** and **Directory.GetFileNames()**. This is quite clumsy, as one cannot use objects to traverse down a tree of directories (as one can do with the **Directory** class), but rather must perform string manipulation to construct paths within the isolated store. Hopefully future versions of the framework will move towards consistency with the **System.IO** namespaces.

Input and output

I/O libraries often use the abstraction of a *stream*, which represents any data source or sink as an object capable of producing or receiving pieces of data. The stream hides the details of what happens to the data inside the actual I/O device.

The C# library classes for I/O are divided by input and output, as you can see by examining the online help reference to the .NET Framework. By inheritance, everything derived from the **Stream** class has basic methods called Read(), ReadByte(), Write(), and WriteByte() for reading and writing arrays and single bytes. However, you won't generally use these methods; they exist so that other classes can use them—these other classes provide a more useful interface. Thus, you'll rarely create an object for input or output by using a single class, but instead will layer multiple objects together to provide your desired functionality. The fact that you create more than one object to create a single resulting stream is the primary reason that .NET's IO library is confusing.

Another sad factor that contributes to confusion is that, alone of all the major .NET namespaces, the **System.IO** namespace violates good object-oriented design principles. In chapter #inheritance#, we spoke of the benefits of aggregating interfaces to specify the mix of abstract data types in an implementation. Rather than do that, the .NET IO classes have an overly-inclusive **Stream** base class and a trio of public instance properties **CanRead**, **CanWrite**, and **CanSeek** that substitute for what should be type information. The motivation for this was probably a well-meaning desire to avoid an "explosion" in types and interfaces, but good design is as simple as possible *and no simpler*. By going too far with **Stream**, and with an unfortunate handful of naming and behavior inconsistencies, the **System.IO** namespace can be quite frustrating.

Types of **Stream**

Classes descended from **Stream** come in two types: implementation classes associated with a particular type of data sink or source:

1. MemoryStreams are the simplest streams and work with in-memory data representations

2. FileStreams work with files and add functions for locking the file for exclusive access. IsolatedStorageFileStream descends from FileStream and is used by isolated stores.

3. NetworkStreams are very helpful when network programming and encapsulate (but provide access to) an underlying network socket.

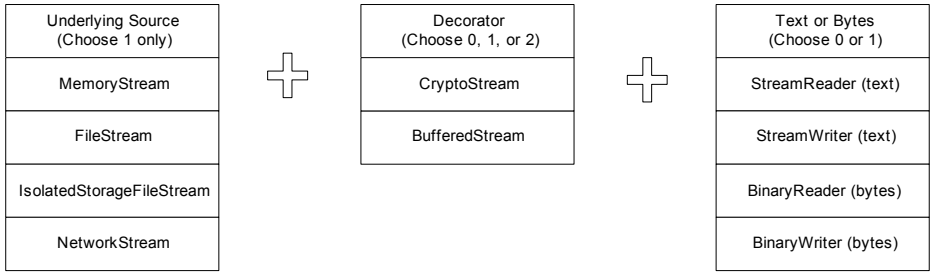
And classes which are used to dynamically add additional responsibilities to other streams:

1. CryptoStreams can encode and decode any other streams, whether those streams originate in memory, the file system, or over a network.
2. BufferedStreams improve the performance of most stream scenarios by reading and writing bytes in large chunks, rather than one at a time.

Classes such as **CryptoStream** and **BufferedStream** are called "Wrapper" or "Decorator" classes (see *Thinking in Patterns*).

Text and Binary

Having determined where the stream is to exist (memory, file, or network) and how it is to be decorated (with cryptography and buffering), you'll need to choose whether you want to deal with the stream as characters or as bytes. If as characters, you can use the **StreamReader** and **StreamWriter** classes to deal with the data as lines of strings, if as bytes, you can use **BinaryReader** and **BinaryWriter** to translate bytes to and from the primitive value types.



All told, there are 90 different valid combinations of these three aspects and while it can be confusing at first, it's clearer than having, for instance, a class called **BinaryCryptoFileReader**. Just to keep you on your toes, though, **StreamReader** and **StreamWriter** have sibling classes **StringReader** and **StringWriter** which work directly on strings, not streams.

Working With Different Sources

This example shows that although the way in which one turns a source into a stream differs, once in hand, any source can be treated equivalently:

```
//:c11:SourceStream.cs
using System;
using System.Text;
using System.IO;

class SourceStream {
   Stream src;

   SourceStream(Stream src){
      this.src = src;
   }


   /*
   static SourceStream ForNetworkStream(){
      string thinkingIn = "www.microsoft.com";
      IPAddress tNet =
Dns.Resolve(thinkingIn).AddressList[0];
      IPEndPoint ep = new IPEndPoint(tNet, 80);
      Socket s = new Socket(
         AddressFamily.InterNetwork,
         SocketType.Stream,
         ProtocolType.Tcp );
      s.Connect(ep);
      NetworkStream nStream =
         new NetworkStream(s);
      SourceStream srcStream =
         new SourceStream(nStream);
      return srcStream;
   }
   */

   void ReadAll(){
      System.Console.WriteLine(
          "Reading stream of type " + src.GetType());
```

```
      int nextByte;
      while ((nextByte = src.ReadByte()) != -1) {
         System.Console.Write((char) nextByte);
      }
   }

   public static void Main(){
      SourceStream srcStr = ForMemoryStream();
      srcStr.ReadAll();
      srcStr = ForFileStream();
      srcStr.ReadAll();
      //srcStr = ForNetworkStream();
      //srcStr.ReadAll();
   }

   static SourceStream ForMemoryStream(){
      string aString = "mary had a little lamb";
      UnicodeEncoding ue = new UnicodeEncoding();
      char bytes = ue.GetBytes(aString);
      MemoryStream memStream =
      new MemoryStream(bytes);
      SourceStream srcStream =
      new SourceStream(memStream);
      return srcStream;
   }

   static SourceStream ForFileStream(){
      string fName = "SourceStream.cs";
      FileStream fStream =
      new FileStream(fName, FileMode.Open);
      SourceStream srcStream =
      new SourceStream(fStream);
      return srcStream;
   }

}
```

The constructor to **SourceStream** takes a **Stream** and assigns it to the instance variable **src**, while the method **ReadAll()** reads that **src** one byte at a time until the method returns -1, indicating that there are no more bytes. Each byte read is cast to a char and sent to the console. The **Main()** method uses the static methods **ForMemoryStream()** and

ForFileStream() to instantiate **SourceStream**s and then calls the **ReadAll()** method.

So far, all the code has dealt with **Stream**s no matter what their real source, but the static methods must necessarily be specific to the subtype of **Stream** being created. In the case of the **MemoryStream**, we start with a **string**, use the **UnicodeEncoding** class from the **System.Text** namespace to convert the **string** into an array of **bytes**, and pass the result into the **MemoryStream** constructor. The **MemoryStream** goes to the **SourceStream** constructor, and then we return the **SourceStream**.

For the FileStream, on the other hand, we have to specify an extant filename and what FileMode we wish to use to open it. The FileMode enumeration includes:

Table 11-1. FileMode Values

FileMode.Value	Behavior
Append	If the file exists, open it and go to the end of the file immediately. If the file does not exist, create it. File cannot be read.
Create	Creates a new file of the given name, *even if* that file already exists (it erases the extant file).
CreateNew	Creates a new file, *if it does not exist*. If the file exists, throws an **IOException**.
Open	Opens an existing file and throws a **FileNotFoundException** otherwise.
OpenOrCreate	Creates and opens a file, creating a new file if necessary
Truncate	Opens an existing file and truncates its size to zero.

Fun With CryptoStreams

Microsoft has done a big favor to eCommerce developers by including industrial-strength cryptography support in the .NET Framework. Many people mistakenly believe that to be anything but a passive consumer of cryptography requires hardcore mathematics. Not so. While few people are capable of developing new fundamental algorithms, cryptographic *protocols* that use the algorithms for complex tasks are accessible to anyone, while actual *applications* that use these protocols to deliver business value are few and far between.

Cryptographic algorithms come in two fundamental flavors: *symmetric algorithms* use the same key to encrypt and decrypt a data stream, while *asymmetric algorithms* have a "public" key for encryption and a "private" key for decryption. The .NET Framework comes with several symmetric algorithms and two asymmetric algorithms, one for general use and one that supports the standard for digital signatures.

Category	Name	Characteristics
Symmetric	DES	Older US Federal standard for "sensitive but not classified" data. 56-bit effective key. Cracked in 22 hours by $250,000 custom computer, plus 100K distributed PCs. If it's worth encrypting, it's worth not using DES.
Symmetric	TripleDES	An extension of DES that has a 112-bit effective key (note that this increases cracking difficulty by 2^{56}).
Symmetric	RC2	Variable key size, implementation seems to be fastest symmetric.

Symmetric	Rijndael	Algorithm chosen for Advanced Encryption Standard, effectively DES replacement. Fast, variable and large key sizes, generally the best symmetric cipher. Pronounced “rain-dahl”
Asymmetric	DSA	Cannot be used for encryption; only good for digital signing.
Asymmetric	RSA	Patent expired, almost synonymous with public-key cryptopgraphy.

CryptoStreams are only created by the symmetric algorithms.

```
//:c11:SecretCode.cs
using System;
using System.IO;
using System.Security.Cryptography;

class SecretCode{
   string fileName;
   string FileName{
      get { return fileName; }
      set { fileName = value; }
   }

   RijndaelManaged rm;

   SecretCode(string fName){
      fileName = fName;
      rm = new RijndaelManaged();
      rm.GenerateKey();
      rm.GenerateIV();
   }

   void EncodeToFile(string outName){
      FileStream src = new FileStream(
         fileName, FileMode.Open);

      ICryptoTransform encoder =
         rm.CreateEncryptor();
```

```
      CryptoStream str = new CryptoStream(
         src, encoder, CryptoStreamMode.Read);

      FileStream outFile = new FileStream(
         outName, FileMode.Create);

      int i = 0;
      while((i = str.ReadByte()) != -1){
         outFile.WriteByte((byte)i);
      }

      src.Close();
      outFile.Close();
   }

   void Decode(string cypherFile){
      FileStream src = new FileStream(
         cypherFile, FileMode.Open);

      ICryptoTransform decoder =
         rm.CreateDecryptor();
      CryptoStream str = new CryptoStream(
         src, decoder, CryptoStreamMode.Read);

      int i = 0;
      while((i = str.ReadByte()) != -1){
         System.Console.Write((char) i);
      }
      src.Close();
   }

   public static void Main(string[] args){
      SecretCode sc = new SecretCode(args[0]);
      sc.EncodeToFile("encoded.dat");
      System.Console.WriteLine("Decoded:");
      sc.Decode("encoded.dat");
   }
}///:~
```

The cryptographic providers are in the System.Security.Cryptography namespace. Each algorithm has both a base class named after the algorithm (DES, Rijndael, RSA, etc.) and an implementation of that

algorithm provided by Microsoft. This is a nice design, allowing one to plug in new implementations of various algorithms as desired.

The System.Security.Cryptography namespace is not part of Microsoft's submission to ECMA and therefore the source code is not available to scrutiny as part of the shared-source Common Language Infrastructure initiative that Microsoft is trying to use to generate goodwill in the academic community. Although Microsoft's implementations have been validated by the US and Canadian Federal governments, it's a pity that this source code is not available for public review.

In this case, we use the RijndaelManaged class that implements the Rijndael algorithm. Like the other implementations, the RijndaelManaged class is able to generate random keys and initialization vectors, as shown in the **SecretCode** constructor, which also sets an instance variable **fileName** to the name of the file which we'll be encrypting.

EncodeToFile() opens a **FileStream** named **src** to our to-be-encrypted file. The symmetric cryptographic algorithms each provide a **CreateEncryptor()** and **CreateDecryptor()** method which returns an **ICryptoTransform** that is a necessary parameter for the **CryptoStream** constructor. With the input stream **src**, the **ICryptoTransform** encoder, and the **CryptoStreamMode.Read** mode as parameters we generate a **CryptoStream** called **str**.

The **outFile** stream is constructed in a familiar way but this time with **FileMode.Create**. We read the **str CryptoStream** and write it to the **outFile**, using the method **WriteByte()**. Once done, we close both the source file and the newly created encrypted file.

The method **Decode()** does the complement; it opens a **FileStream**, uses the **RijndaelManaged** instance to create an **ICryptoTransform decoder** and a **CryptoStream** appropriate for reading the encrypted file. We read the encrypted file one byte at a time and print the output on the console.

The **Main()** method creates a new **SecretCode** class, passing in the first command-line argument as the filename to be encoded. Then, the call to

EncodeToFile() encrypts that file to another called "encoded.dat." Once that file is created, it is in turn decoded by the **Decode()** method.

One characteristic of a good encrypted stream is that it is difficult to distinguish from a stream of random data; since random data is non-compressible, if you attempt to compress "encoded.dat" you should see tht sure enough the "compressed" file is larger than the original.

BinaryReader and BinaryWriter

While we've been able to get by with reading and writing individual bytes, doing so requires a lot of extra effort when dealing with anything but the simplest data. **BinaryReader** and **BinaryWriter** are wrapper classes which can ease the task of dealing with the most common primitive value types. The BinaryWriter class contains a large number of overridden **Write()** methods, as illustrated in this sample:

```
//:c11:BinaryWrite.cs
using System;
using System.IO;

class BinaryWrite{
   public static void Main(){
      Stream fStream = new FileStream(
         "binaryio.dat", FileMode.Create);
      WriteTypes(fStream);
      fStream.Close();
   }

   static void WriteTypes(Stream sink){
      BinaryWriter bw = new BinaryWriter(sink);

      bw.Write(true);
      bw.Write(false);
      bw.Write((byte) 7);
      bw.Write(new byte[] { 1, 2, 3, 4 });
      bw.Write('z');
      bw.Write(new char[] { 'A', 'B', 'C', 'D' });
      bw.Write(new Decimal(123.45));
      bw.Write(123.45);
      bw.Write((short) 212);
```

```
      bw.Write((long) 212);
      bw.Write("<boolean>true</boolean>");
   }

}///:~
```

BinaryWrite's Main() method creates a FileStream for writing, upcasts the result to **Stream**, passes it to the static **WriteTypes()** method, and afterwards closes it. The **WriteTypes()** method takes the passed in **Stream** and passes it as a parameter to the **BinaryWriter** constructor. Then, we call **BinaryWriter.Write()** with various parameters, everything from **bool** to **string**. Behind the scenes, the **BinaryWriter** turns these types into sequences of bytes and writes them to the underlying stream.

Every type, except for **string**, has a predetermined length in bytes – even bools, which could be represented in a single bit, are stored as a full byte --- so it might be more accurate to call this type of storage "byte data" rather than "binary data." To store a string, **BinaryWriter** first writes one or more bytes to indicate the number of bytes that the string requires for storage; these bytes use 7 bits to encode the length and the 8th bit (if necessary) to indicate that the next byte is not the first character of the string, but another length byte.

The **BinaryWriter** class does nothing we couldn't do on our own, but it's much more convenient. Naturally, there's a complementary **BinaryReader** class, but because one cannot have polymorphism based only on return type (see chapter #polymorphism#), the methods for reading various types are a little longer:

```
//:c11:BinaryRead.cs
using System;
using System.IO;

class BinaryRead{
   public static void Main(string[] args){
      Stream fStream = new BufferedStream(
          new FileStream(args[0], FileMode.Open));
      ByteDump(fStream);
      fStream.Close();
      fStream = new BufferedStream(
```

```
            new FileStream(args[0], FileMode.Open));
        ReadTypes(fStream);
        fStream.Close();
    }

    static void ByteDump(Stream src){
        int i = 0;
        while((i = src.ReadByte()) != -1){
            System.Console.WriteLine(
                "{0} = {1} ", (char) i, i);
        }
        System.Console.WriteLine();
    }

    static void ReadTypes(Stream src){
        BinaryReader br = new BinaryReader(src);

        bool b = br.ReadBoolean();
        System.Console.WriteLine(b);
        b = br.ReadBoolean();
        System.Console.WriteLine(b);
        byte bt = br.ReadByte();
        System.Console.WriteLine(bt);
        byte[] byteArray = br.ReadBytes(4);
        System.Console.WriteLine(byteArray);
        char c = br.ReadChar();
        System.Console.WriteLine(c);
        char[] charArray = br.ReadChars(4);
        System.Console.WriteLine(charArray);
        Decimal d = br.ReadDecimal();
        System.Console.WriteLine(d);
        Double db = br.ReadDouble();
        System.Console.WriteLine(db);
        short s = br.ReadInt16();
        System.Console.WriteLine(s);
        long l = br.ReadInt64();
        System.Console.WriteLine(l);
        string tag = br.ReadString();
        System.Console.WriteLine(tag);
    }
}///:~
```

BinaryRead.Main() introduces another wrapper class, **BufferedStream**, which increases the efficiency of non-memory-based streams by using an internal memory buffer to temporarily store the data rather than writing a single byte to the underlying file or network. BufferedStreams are largely transparent to use, although the method **Flush()**, which sends the contents of the buffer to the underlying stream, regardless of whether it's full or not, can be used to fine-tune behavior.

BinaryRead works on a file whose name is passed in on the command line. **ByteDump()** shows the contents of the file on the console, printing the byte as both a character and displaying its decimal value. When run on "binaryio.dat", the run begins:

```
☺ = 1
  = 0
 = 7
☺ = 1
☻ = 2
♥ = 3
♦ = 4
z = 122
A = 65
B = 66
C = 67
D = 68
…etc…
```

The first two bytes represent the Boolean values true and false, while the next parts of the file correspond directly to the values of the bytes and chars we wrote with the program **BinaryWrite**. The more complicated data types are harder to interpret, but towards the end of this method, you'll see a byte value of 212 that corresponds to the **short** and the **long** we wrote.

The last part of the output from this method looks like this:

```
↕ = 23
< = 60
b = 98
o = 111
o = 111
l = 108
```

```
e = 101
a = 97
n = 110
> = 62
t = 116
r = 114
u = 117
e = 101
< = 60
/ = 47
b = 98
o = 111
o = 111
l = 108
e = 101
a = 97
n = 110
> = 62
```

This particular string, which consumes 24 bytes of storage (1 length byte, and 23 character bytes), is the XML equivalent of the single byte at the beginning of the file that stores a **bool**. We'll discuss XML in length in chapter #XML#, but this shows the primary trade-off between binary data and XML – efficiency versus descriptiveness. Ironically, while local storage is experiencing greater-than-Moore's-Law increases in data density (and thereby becoming cheaper and cheaper) and network bandwidth (especially to the home and over wireless) will be a problem for the foreseeable future, file formats remain primarily binary and XML is exploding as the over-network format of choice!

After **BinaryRead** dumps the raw data to the console, it then reads the same stream, this time with the static method **ReadTypes()**. **ReadTypes()** instantiates a **BinaryReader()** and calls it's various **Readxxx()** methods in exact correspondence to the **BinaryWriter.Write()** methods of **BinaryWrite.WriteTypes()**. When run on binaryio.dat, **BinaryRead.ReadTypes()** reproduces the exact data, but you can also run the program on any file and it will gamely interpret that program's bytes as the specified types. Here's the output when **BinaryRead** is run on its own source file:

```
True
```

```
True
58
System.Byte[]
B
inar
-3.5732267922136636517188457081E-75
6.2763486340252E-245
29962
8320773185183050099
em.IO;

class BinaryRead{
   public static void Main(string[] args){
      Stream fStream = new BufferedStream(
```

Again, this is the price to be paid for the efficiency of byte data – the slightest discrepancy between the types specified when the data is written and when it is read leads to incorrect data values, but the problem will probably not be detected until some *other* method attempts to use this wrong data.

StreamReader and StreamWriter

Because strings are such a common data type, the .NET Framework provides some decorator classes to aid in reading lines and blocks of text. The **StreamReader** and **StreamWriter** classes decorate streams and **StringReader** and **StringWriter** decorate **string**s.

The most useful method in **StreamReader** is **ReadLine()**, as demonstrated in this sample, which prints a file to the console with line numbers prepended:

```
//:c11:LineAtATime.cs
using System;
using System.IO;

class LineAtATime{
   public static void Main(string[] args){
      foreach(string fName in args){
         Stream src = new BufferedStream(
            new FileStream(fName, FileMode.Open));
         LinePrint(src);
```

```
            src.Close();
        }
    }

    static void LinePrint(Stream src){
        StreamReader r = new StreamReader(src);
        int line = 0;
        string aLine = "";
        while((aLine = r.ReadLine()) != null){
            System.Console.WriteLine("{0}: {1}",
                line++, aLine);
        }
    }
}///:~
```

The **Main()** method takes a command-line filename and opens it, decorates the **FileStream** with a **BufferedStream**, and passes the resulting **Stream** to the **LinePrint()** static method. **LinePrint()** creates a new **StreamReader** to decorate the **BufferedStream** and uses **StreamReader.ReadLine()** to read the underlying stream a line at a time. **StreamReader.ReadLine()** returns a null reference at the end of the file, ending the output loop.

StreamReader is useful for writing lines and blocks of text, and contains a slew of overloaded **Write()** methods similar to those in BinaryWriter, as this example shows:

```
//:c11:TextWrite.cs
using System;
using System.IO;

class TextWrite{
    public static void Main(){
        Stream fStream = new FileStream(
            "textio.dat", FileMode.Create);
        WriteLineTypes(fStream);
        fStream.Close();
    }

    static void WriteLineTypes(Stream sink){
        StreamWriter sw = new StreamWriter(sink);
```

```
        sw.WriteLine(true);
        sw.WriteLine(false);
        sw.WriteLine((byte) 7);
        sw.WriteLine('z');
        sw.WriteLine(new char[] { 'A', 'B', 'C', 'D' });
        sw.WriteLine(new Decimal(123.45));
        sw.WriteLine(123.45);
        sw.WriteLine((short) 212);
        sw.WriteLine((long) 212);
        sw.WriteLine("{0} : {1}",
              "string formatting supported", "true");

        sw.Close();
    }

}///:~
```

Like the BinaryWrite sample, this program creates a filestream (this time for a file called "textio.dat") and passes that to another method that decorates the underlying data sink and writes to it. In addition to **Write()** methods that are overloaded to write the primitive types, **StreamWriter** will call **ToString()** on *any* object and supports string formatting. In one of the namespace's annoyances, **StreamWriter** is buffered (although it doesn't descend from **BufferedStream**), and so you must explicitly call **Close()** in order to flush the lines to the underlying stream.

The data written by **StreamWriter** is in text format, as shown in the contents of textio.dat:

```
True
False
7
z
ABCD
123.45
123.45
212
212
string formatting supported : true
```

Bear in mind that **StreamReader** does not have **Readxxx()** methods – if you want to store primitive types to be read and used as primitive types, you should use the byte-oriented **Reader** and **Writer** classes. You *could*

store the data as text, read it as text, and then perform the various string parsing operations to recreate the values, but that would be wasteful.

It's worth noting that **StreamReader** and **StreamWriter** have sibling classes **StringReader** and **StringWriter** that are descended from the same **Reader** and **Writer** abstraction. Since **string** objects are immutable (once set, a **string** cannot be changed), there is a need for an efficient tool for building strings and complex formatting tasks. The basic task of building a string from substrings is handled by the **StringBuilder** class, while the complex formatting can be done with the **StringWriter** (which decorates a **StringBuilder** in the same way that the **StreamWriter** decorates a **Stream**).

Random access with Seek

The Stream base class contains a method called **Seek()** that can be used to jump between records and data sections of known size (or sizes that can be computed by reading header data in the stream). The records don't have to be the same size; you just have to be able to determine how big they are and where they are placed in the file. The Seek() method takes a long (implying a maximum file size of 8 exabytes, which will hopefully suffice for a few years) and a value from the **SeekOrigin** enumeration which can be **Begin**, **Current**, or **End**. The SeekOrigin value specifies the point from which the seek jumps.

Although **Seek()** is defined in **Stream**, not all **Stream**s support it (for instance, one can't "jump around" a network stream). The **CanSeek** bool property specifies whether the stream supports **Seek()** and the related **Length()** and **SetLength()** mehods, as well as the **Position()** method which returns the current position in the **Stream**. If **CanSeek** is **false** and one of these methods is called, it will throw a **NotSupportedException**. This is poor design. Support for random access is based on type, not state, and should be specified in an interface (say, **ISeekable**) that is implemented by the appropriate subtypes of **Stream**.

If you use **SeekOrigin.End**, you should use a negative number for the offset; performing a **Seek()** beyond the end of the **stream** moves to the end of the file (i.e., **ReadByte()** will return a -1, etc.).

This example shows the basic use of **Stream.Seek()**:

```
//:c11:FibSeek.cs
using System;
using System.IO;

class FibSeek {
   Stream src;

   FibSeek(Stream src){
      this.src = src;
   }

   void DoSeek(SeekOrigin so){
      if (so == SeekOrigin.End) {
         src.Seek(-10, so);
      } else {
         src.Seek(10, so);
      }
      int i = src.ReadByte();
      System.Console.WriteLine(
          "10 bytes from {0} is : {1}", so, (char) i);
   }

   public static void Main(string[] args){
      foreach(string fName in args){
         FileStream f = null;
         try {
            f = new FileStream(fName, FileMode.Open);
            FibSeek fs = new FibSeek(f);
            fs.DoSeek(SeekOrigin.Begin);
            fs.DoSeek(SeekOrigin.End);
            f.Seek(12, SeekOrigin.Begin);
            fs.DoSeek(SeekOrigin.Current);
         } catch (Exception ex) {
            System.Console.WriteLine(ex);
         } finally {
            f.Close();
         }
      }
   }
```

```
}///:~
```

Standard I/O

The term *standard I/O* refers to the Unix concept (which is reproduced in some form in Windows and many other operating systems) of a single stream of information that is used by a program. All the program's input can come from *standard input*, all its output can go to *standard output*, and all of its error messages can be sent to *standard error*. The value of standard I/O is that programs can easily be chained together and one program's standard output can become the standard input for another program. More than just a convenience, this is a powerful architectural pattern called *Pipes and Filters*; although this architecture was not very common in the 1990s, it's a very powerful one, as anyone who's witnessed a UNIX guru can testify.

Reading from standard input

Following the standard I/O model, the Console class exposes three static properties: Out, Error, and In. The System class has a static Console property that we've been using for output throughout the book and in Chapter #Exception# we sent some error messages to System.Console.Error. Out and Error are TextWriters, while In is a TextReader.

Typically, you either want to read console input as either a character or a complete line at a time. Here's an example that simply echoes each line that you type in:

```
//c11:EchoIn.cs
// How to read from standard input.

public class EchoIn {
  public static void Main(){
    string s;
    while((s = System.Console.In.ReadLine()).Length
           != 0)
      System.Console.WriteLine(s);
    // An empty line terminates the program
  }
```

```
} ///:~
```

Redirecting standard I/O

The **Console** class allows you to redirect the standard input, output, and error I/O streams using simple static method calls:

SetIn(TextReader)
SetOut(TextWriter)
SetError(TextWriter)

(There is no obvious reason why these methods are used rather than allowing the Properties to be set directly.)

Redirecting output is especially useful if you suddenly start creating a large amount of output on your screen and it's scrolling past faster than you can read it. Redirecting input is valuable for a command-line program in which you want to test a particular user-input sequence repeatedly. Here's a simple example that shows the use of these methods:

```
//: c11:Redirecting.cs
// Demonstrates standard I/O redirection.
using System;
using System.IO;

public class Redirecting {
  public static void Main(){
    StreamReader sr = new StreamReader(
        new BufferedStream(
           new FileStream(
              "Redirecting.cs", FileMode.Open)));
    StreamWriter sw = new StreamWriter(
        new BufferedStream(
           new FileStream(
              "redirect.dat", FileMode.Create)));
    System.Console.SetIn(sr);
    System.Console.SetOut(sw);
    System.Console.SetError(sw);


    String s;
```

```
    while((s = System.Console.In.ReadLine()) != null)
      System.Console.Out.WriteLine(s);
    System.Console.Out.Close(); // Remember this!
  }
} ///:~
```

This program attaches standard input to a file, and redirects standard output and standard error to another file.

Regular Expressions

Regular expressions are a powerful pattern-matching tool for interpreting and manipulating strings. Although regular expressions are not necessarily related to input and output, it is probably their most common application, so we'll discuss them here.

Regular expressions have a long history in the field of computer science but continue to be expanded and improved, which gives rise to an intimidating set of capabilities and alternate routes to a given end. The regular expressions in the .NET Framework are Perl 5 compatible but include additional features such as right-to-left matching and do not require a separate compilation step.

The fundamental responsibility of the **System.Text.RegularExpressions Regex** class is to match a given pattern with a given target string. The pattern is described in a terse notation that combines literal text that must appear in the target with meta-text that specifies both acceptable variations in text and desired manipulations such as variable assignment or text replacement.

This sample prints out the file names and lines that match a regular expression typed in the command line:

```
//:c11:TGrep.cs
//Demonstrate basic regex matching against files
using System;
using System.IO;
using System.Text.RegularExpressions;

class TGrep{
   public static void Main(string[] args){
```

```
      TGrep tg = new TGrep(args[0]);
      tg.ApplyToFiles(args[1]);
   }
   Regex re;

   TGrep(string pattern){
       re = new Regex(pattern);
   }

   void ApplyToFiles(string fPattern){
       string[] fNames =
         Directory.GetFiles(".", fPattern);
       foreach (string fName in fNames ) {
          StreamReader sr = null;
          try{
            sr = new StreamReader(
              new BufferedStream(
                 new FileStream(
                    fName, FileMode.Open)));
            string line = "";
            int lCount = 0;
            while((line = sr.ReadLine()) != null){
               lCount++;
               if(re.IsMatch(line)){
                System.Console.WriteLine(
                 "{0} {1}: {2}", fName, lCount, line);
               }
            }
           }finally{
              sr.Close();
           }
       }
   }
}///:~
```

The **Main()** method passes the first command-line argument to the **TGrep()** constructor, which in turn passes it to the **Regex()** constructor. The second argument is then passed as the argument to the **ApplyToFiles()** method.

ApplyToFiles() uses IO techniques we've discussed previously to read a series of files line-by-line and incrementing the variable **lCount** to let us

know what line number works. Each line is passed to the **Regex.IsMatch()** method, and if that method returns **true**, the filename, line number, and contents of the line are printed to the screen.

You might guess that "tgrep using tgrep.cs" would print lines 3, 4, and 5 of tgrep.cs, but you might not expect that "tgrep [0-9] tgrep.cs" would print every line that contains a number, or that "tgrep [\s]f[\w]*[\s]*= *.cs" would print every line that assigns a value to a variable that begins with a lowercase 'f'. Like SQL in ADO.NET, the regular expression notation is a separate language quite unlike C#, and *Thinking in Regular Expressions* would be quite a different book than this one.

In addition to simply determining if a match exists, **Regex** can actually return the value of the matches, as this program demonstrates:

```
//:c11:GrepMatches.cs

using System;
using System.IO;
using System.Text.RegularExpressions;

class GrepMatches{

   public static void Main(string[] args){
      GrepMatches tg = new GrepMatches(args[0]);
      string target = args[1];
      tg.ApplyToFiles(target);
   }

   Regex re;

   GrepMatches(string pattern){
       re = new Regex(pattern);
   }

   void ApplyToFiles(string fPattern){
       string[] fNames = Directory.GetFiles(
         ".", fPattern);
       foreach (string fName in fNames ) {
         StreamReader sr = null;
```

```
          try{
           sr = new StreamReader(
             new BufferedStream(
               new FileStream(fName, FileMode.Open)));
           string line = "";
           int lCount = 0;
           while((line = sr.ReadLine()) != null){
              lCount++;
              if(re.IsMatch(line)){
                System.Console.WriteLine(
                  "{0} {1}: {2}", fName, lCount, line);
                ShowMatches(re.Matches(line));
              }
           }
         }finally{
           sr.Close();
         }
      }
   }

   private void ShowMatches(MatchCollection mc){
      for(int i = 0; i < mc.Count; i++){
         System.Console.WriteLine(
            "Match[{0}] = {1}", i, mc[i]);
      }
   }
}
```

Regex.Matches() returns a **MatchCollection** which naturally contains **Match** objects. This sample program can be helpful in debugging the development of a regular expression, which for most of us requires a considerable amount of trial and error!

The static method Regex.Replace() can make complex transformations surprisingly straightforward. This sample makes pattern substitutions in a text file:

```
//:c11:TSed.cs
using System;
using System.IO;
using System.Text.RegularExpressions;
```

```
class TSed{
   public static void Main(string[] args){
      TSed tg = new TSed(args[0], args[1]);
      string target = args[2];
      tg.ApplyToFiles(target);
   }

   string pattern;
   string rep;

   TSed(string pattern, string rep){
       this.pattern = pattern;
       this.rep = rep;
   }

   void ApplyToFiles(string fPattern){
       string[] fNames =
          Directory.GetFiles(".", fPattern);
       foreach (string fName in fNames ) {
          StreamReader sr = null;
          try{
            sr = new StreamReader(
              new BufferedStream(
                 new FileStream(
                   fName, FileMode.Open)));
            string line = "";
            int lCount = 0;
            while((line = sr.ReadLine()) != null){
               string nLine =
                  Regex.Replace(line, pattern, rep);
               System.Console.WriteLine(nLine);
            }
           }finally{
              sr.Close();
            }
       }
   }
}
```

Like the previous samples, this one works with command-line arguments, but this time, instead of instantiating a **Regex** for pattern-matching, the first two command-line arguments are just stored as strings, which are

later passed to the **Regex.Replace()** method. If the pattern matches, the replacement pattern is inserted into the string, if not, the line is untouched. Whether touched or not, the line is written to the console; this makes this program a "tiny" version of UNIX's sed command and is very convenient.

Checking capitalization style

In this section we'll look at a complete example of the use of C# IO which also uses regular expression. This project is directly useful because it performs a style check to make sure that your capitalization conforms to the C# style. It opens each **.cs** file in the current directory and extracts all the class names and identifiers, then shows you if any of them don't meet the C# style. You can then use the TSed sample above to automatically replace them.

The program uses two regular expressions that match words that preced a block and which begin with a lowercase letter. One **Regex** matches block-oriented identifiers (such as class, interface, property, and namespace names) and the other catches method declarations. Doing this in a single **Regex** is one of the exercises at the end of the chapter.

```
//:c11:CapStyle.cs
//Scans all .cs files for properly capitalized
//method and classnames
using System;
using System.IO;
using System.Text.RegularExpressions;

public class CapStyle{
   public static void Main(){
      string[] fNames =
         Directory.GetFiles(".","*.cs");
      foreach(string fName in fNames){
         CapStyle cs = null;
         try{
            cs = new CapStyle(fName);
            cs.Check();
         }finally{
            cs.Close();
```

```
      }
   }
}

string[] keyWords= new string[]{
   "abstract", "event", "new", "struct", "as",
   "explicit", "null", "switch", "base", "extern",
   "object", "this", "bool", "false", "operator",
   "throw", "break", "finally", "out", "true",
   "byte", "fixed", "override", "try", "case",
   "float", "params", "typeof", "catch", "for",
   "private", "uint", "char", "foreach",
   "protected", "ulong", "checked", "goto",
   "public", "unchecked", "class", "if",
   "readonly", "unsafe", "const", "implicit",
   "ref", "ushort", "continue", "in", "return",
   "using", "decimal", "int", "sbyte", "virtual",
   "default", "interface", "sealed", "volatile",
   "delegate", "internal", "short", "void", "do",
   "is", "sizeof", "while", "double", "lock",
   "stackalloc", "else", "long", "static", "enum",
   "namespace", "string", "try", "catch",
   "finally", "using", "else", "switch", "public",
   "static", "void", "foreach", "if", "while",
   "bool", "byte", "for", "get", "set"
};

StreamReader fStream;

Regex blockPrefix;
Regex methodDef;

CapStyle(string fName){
   fStream = new StreamReader(
      new BufferedStream(
         new FileStream(fName, FileMode.Open)));
   /*
   matches just-before-bracket identifier
   starting with lowercase
   */
   blockPrefix =
```

```
            new Regex(@"[\s](?<id>[a-z][\w]*)[\s]*{");

        /*
        matches just-before-bracket with argument list
        and identifier starting with lowerCase
        */
        methodDef =
            new Regex(
              @"[\s](?<id>[a-z][\w]*)\s*\((.*)\)[\s]*{");

        System.Console.WriteLine(
            "Checking file: " + fName);
    }

    void Close(){
        fStream.Close();
    }

    void Check(){
        string line = "";
        int lCount = 0;
        while((line = fStream.ReadLine()) != null){
            lCount++;
            if(Suspicious(line)){
                System.Console.WriteLine(
                    "{0}: {1}", lCount, line);
            }
        }
    }

    bool Suspicious(string line){
        if(MatchNotKeyword(line, blockPrefix) == true){
            return true;
        }
        if(MatchNotKeyword(line, methodDef) == true){
            return true;
        }
        return false;
    }

    bool MatchNotKeyword(string line, Regex re){
```

```
      if(re.IsMatch(line)){
         Match m = re.Match(line);
         string identifier = m.Groups["id"].Value;
         if(Array.IndexOf(keyWords, identifier) < 0){
            return true;
         }
      }
      return false;
   }
}
```

The **Main()** generates a list of all the C# files in the current directory and for each one creates a **CapStyle** instance, runs **CapStyle.Check()** and then **CapStyle.Close().**

Each **CapStyle** instance contains a list of C# keywords that are allowed to be in lowercase, as well as instance variables that hold the two regular expressions, and a **StreamReader** instance variable that reads the underlying file. The **CapStyle()** constructor opens the file and constructs the two two regular expressions. The expressions will match namespaces, class and interface identifiers, properties, and method names that precede a '{' character (handling multi-line bracketing conventions is another exercise!). Additionally, the expressions use *group naming* to associate the word that begins with a lowercase letter to a regex variable called **id** ("`(?<id>[a-z][\w]*)`" is the relevant notation; the parentheses specify the group, the `?<id>` specifies the name).

The **Check()** method goes through the **StreamReader** line-by-line, seeing if **Suspicious()** returns true; if so, that line is output to the Console. **Suspicious()** in turn calls **MatchNotKeyword()**, passing in the suspect line and a reference to one of the two instance **Regex**s. **MatchNotKeyword()** checks for a match; if there is one it assigns the value of the **Regex** group named **id** to the string **identifier**. If this string does not appear in the array of C# keywords, **MatchNotKeyword()** returns **true**, which causes **Suspicious** to return **true** to **Check()**.

In addition to not handling multi-line bracketing, this program sometimes marks **string**s that contain formatting brackets incorrectly. If you improve the program, please drop the authors a line at www.thinkingin.net.

Summary

The .NET IO stream library does satisfy the basic requirements: you can perform reading and writing with the console, a file, a block of memory, or even across the Internet (as you will see in Chapter #networking#). With inheritance, you can create new types of input and output objects.

The IO library brings up mixed feelings; it does the job and it uses the Decorator pattern to good effect. But if you don't already understand the Decorator pattern, the design is nonintuitive, so there's extra overhead in learning and teaching it. There are also some poor choices in naming and implementation issues.

However, once you *do* understand the fundamentals of **Stream**s and the Decorator pattern and begin using the library in situations that require its flexibility, you can begin to benefit from this design, at which point its cost in extra lines of code will not bother you at all.

Exercises

12: Reflection and Attributes

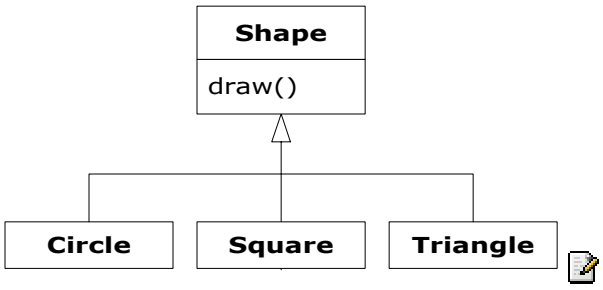
The idea of run-time type identification (RTTI) seems fairly simple at first: it lets you find the exact type of an object when you only have a reference to the base type.

However, the *need* for RTTI uncovers a whole plethora of interesting (and often perplexing) OO design issues, and raises fundamental questions of how you should structure your programs.

This chapter looks at the ways that Java allows you to discover information about objects and classes at run-time. This takes two forms: "traditional" RTTI, which assumes that you have all the types available at compile-time and run-time, and the "reflection" mechanism, which allows you to discover class information solely at run-time. The "traditional" RTTI will be covered first, followed by a discussion of reflection.

The need for RTTI

Consider the now familiar example of a class hierarchy that uses polymorphism. The generic type is the base class **Shape**, and the specific derived types are **Circle**, **Square**, and **Triangle**:



This is a typical class hierarchy diagram, with the base class at the top and the derived classes growing downward. The normal goal in object-oriented programming is for the bulk of your code to manipulate references to the base type (**Shape**, in this case), so if you decide to extend the program by adding a new class (**Rhomboid**, derived from **Shape**, for example), the bulk of the code is not affected. In this example, the dynamically bound method in the **Shape** interface is **draw( )**, so the intent is for the client programmer to call **draw( )** through a generic **Shape** reference. **draw( )** is overridden in all of the derived classes, and because it is a dynamically bound method, the proper behavior will occur even though it is called through a generic **Shape** reference. That's polymorphism.

Thus, you generally create a specific object (**Circle**, **Square**, or **Triangle**), upcast it to a **Shape** (forgetting the specific type of the object), and use that anonymous **Shape** reference in the rest of the program.

As a brief review of polymorphism and upcasting, you might code the above example as follows:

```
//: c12:Shapes.java
import java.util.*;

class Shape {
  void draw() {
    System.out.println(this + ".draw()");
  }
}

class Circle extends Shape {
  public String toString() { return "Circle"; }
}

class Square extends Shape {
  public String toString() { return "Square"; }
}

class Triangle extends Shape {
  public String toString() { return "Triangle"; }
```

```
}

public class Shapes {
  public static void main(String[] args) {
    List s = new ArrayList();
    s.add(new Circle());
    s.add(new Square());
    s.add(new Triangle());
    Iterator e = s.iterator();
    while(e.hasNext())
      ((Shape)e.next()).draw();
  }
} ///:~
```

The base class contains a **draw( )** method that indirectly uses **toString( )** to print an identifier for the class by passing **this** to **System.out.println( )**. If that function sees an object, it automatically calls the **toString( )** method to produce a **String** representation.

Each of the derived classes overrides the **toString( )** method (from **object**) so that **draw( )** ends up printing something different in each case. In **main( )**, specific types of **Shape** are created and then added to a **List**. This is the point at which the upcast occurs because the **List** holds only **object**s. Since everything in Java (with the exception of primitives) is an **object**, a **List** can also hold **Shape** objects. But during an upcast to **object**, it also loses any specific information, including the fact that the objects are **Shape**s. To the **ArrayList**, they are just **object**s.

At the point you fetch an element out of the **List** with **next( )**, things get a little busy. Since the **List** holds only **object**s, **next( )** naturally produces an **object reference**. But we know it's really a **Shape** reference, and we want to send **Shape** messages to that object. So a cast to **Shape** is necessary using the traditional "**(Shape)**" cast. This is the most basic form of RTTI, since in Java all casts are checked at run-time for correctness. That's exactly what RTTI means: at run-time, the type of an object is identified.

In this case, the RTTI cast is only partial: the **object** is cast to a **Shape**, and not all the way to a **Circle**, **Square**, or **Triangle**. That's because the only thing we *know* at this point is that the **List** is full of **Shape**s. At

compile-time, this is enforced only by your own self-imposed rules, but at run-time the cast ensures it.

Now polymorphism takes over and the exact method that's called for the **Shape** is determined by whether the reference is for a **Circle**, **Square**, or **Triangle**. And in general, this is how it should be; you want the bulk of your code to know as little as possible about *specific* types of objects, and to just deal with the general representation of a family of objects (in this case, **Shape**). As a result, your code will be easier to write, read, and maintain, and your designs will be easier to implement, understand, and change. So polymorphism is the general goal in object-oriented programming.

But what if you have a special programming problem that's easiest to solve if you know the exact type of a generic reference? For example, suppose you want to allow your users to highlight all the shapes of any particular type by turning them purple. This way, they can find all the triangles on the screen by highlighting them. Or perhaps your method needs to "rotate" a list of shapes, but it makes no sense to rotate a circle so you'd like to skip only the circle objects. This is what RTTI accomplishes: you can ask a **Shape** reference the exact type that it's referring to. With RTTI you can select and isolate special cases.

The **Class** object

To understand how RTTI works in Java, you must first know how type information is represented at run-time. This is accomplished through a special kind of object called the *Class object*, which contains information about the class. (This is sometimes called a *meta-class*.) In fact, the **Class** object is used to create all of the "regular" objects of your class.

There's a **Class** object for each class that is part of your program. That is, each time you write and compile a new class, a single **Class** object is also created (and stored, appropriately enough, in an identically named **.class** file). At run-time, when you want to make an object of that class, the Java Virtual Machine (JVM) that's executing your program first checks to see if the **Class** object for that type is loaded. If not, the JVM loads it by finding the **.class** file with that name. Thus, a Java program isn't completely

loaded before it begins, which is different from many traditional languages.

Once the **Class** object for that type is in memory, it is used to create all objects of that type.

If this seems shadowy or if you don't really believe it, here's a demonstration program to prove it:

```
//: c12:SweetShop.java
// Examination of the way the class loader works.

class Candy {
  static {
    System.out.println("Loading Candy");
  }
}

class Gum {
  static {
    System.out.println("Loading Gum");
  }
}

class Cookie {
  static {
    System.out.println("Loading Cookie");
  }
}

public class SweetShop {
  public static void main(String[] args) {
    System.out.println("inside main");
    new Candy();
    System.out.println("After creating Candy");
    try {
      Class.forName("Gum");
    } catch(ClassNotFoundException e) {
      e.printStackTrace(System.err);
    }
    System.out.println(
```

```
      "After Class.forName(\"Gum\")");
    new Cookie();
    System.out.println("After creating Cookie");
  }
} ///:~
```

Each of the classes **Candy**, **Gum**, and **Cookie** have a **static** clause that is executed as the class is loaded for the first time. Information will be printed to tell you when loading occurs for that class. In **main( )**, the object creations are spread out between print statements to help detect the time of loading.

A particularly interesting line is:

```
Class.forName("Gum");
```

This method is a **static** member of **Class** (to which all **Class** objects belong). A **Class** object is like any other object and so you can get and manipulate a reference to it. (That's what the loader does.) One of the ways to get a reference to the **Class** object is **forName( )**, which takes a **String** containing the textual name (watch the spelling and capitalization!) of the particular class you want a reference for. It returns a **Class** reference.

The output of this program for one JVM is:

```
inside main
Loading Candy
After creating Candy
Loading Gum
After Class.forName("Gum")
Loading Cookie
After creating Cookie
```

You can see that each **Class** object is loaded only when it's needed, and the **static** initialization is performed upon class loading.

Class literals

Java provides a second way to produce the reference to the **Class** object, using a *class literal*. In the above program this would look like:

```
Gum.class;
```

which is not only simpler, but also safer since it's checked at compile-time. Because it eliminates the method call, it's also more efficient.

Class literals work with regular classes as well as interfaces, arrays, and primitive types. In addition, there's a standard field called **TYPE** that exists for each of the primitive wrapper classes. The **TYPE** field produces a reference to the **Class** object for the associated primitive type, such that:

... is equivalent to ...	
boolean.class	**Boolean.TYPE**
char.class	**Character.TYPE**
byte.class	**Byte.TYPE**
short.class	**Short.TYPE**
int.class	**Integer.TYPE**
long.class	**Long.TYPE**
float.class	**Float.TYPE**
double.class	**Double.TYPE**
void.class	**Void.TYPE**

My preference is to use the "**.class**" versions if you can, since they're more consistent with regular classes.

Checking before a cast

So far, you've seen RTTI forms including:

3. The classic cast; e.g., "**(Shape),**" which uses RTTI to make sure the cast is correct and throws a **ClassCastException** if you've performed a bad cast.

16. The **Class** object representing the type of your object. The **Class** object can be queried for useful run-time information.

In C++, the classic cast "**(Shape)**" does *not* perform RTTI. It simply tells the compiler to treat the object as the new type. In Java, which does perform the type check, this cast is often called a "type safe downcast." The reason for the term "downcast" is the historical arrangement of the class hierarchy diagram. If casting a **Circle** to a **Shape** is an upcast, then casting a **Shape** to a **Circle** is a downcast. However, you know a **Circle** is also a **Shape**, and the compiler freely allows an upcast assignment, but you *don't* know that a **Shape** is necessarily a **Circle**, so the compiler doesn't allow you to perform a downcast assignment without using an explicit cast.

There's a third form of RTTI in Java. This is the keyword **instanceof** that tells you if an object is an instance of a particular type. It returns a **boolean** so you use it in the form of a question, like this:

```
if(x instanceof Dog)
  ((Dog)x).bark();
```

The above **if** statement checks to see if the object **x** belongs to the class **Dog** *before* casting **x** to a **Dog**. It's important to use **instanceof** before a downcast when you don't have other information that tells you the type of the object; otherwise you'll end up with a **ClassCastException**.

Ordinarily, you might be hunting for one type (triangles to turn purple, for example), but you can easily tally *all* of the objects using **instanceof**. Suppose you have a family of **Pet** classes:

```
//: c12:Pets.java
class Pet {}
class Dog extends Pet {}
class Pug extends Dog {}
class Cat extends Pet {}
class Rodent extends Pet {}
class Gerbil extends Rodent {}
class Hamster extends Rodent {}

class Counter { int i; } ///:~
```

The **Counter** class is used to keep track of the number of any particular type of **Pet**. You could think of it as an **Integer** that can be modified.

Using **instanceof**, all the pets can be counted:

```
//: c12:PetCount.java
// Using instanceof.
import java.util.*;

public class PetCount {
  static String[] typenames = {
    "Pet", "Dog", "Pug", "Cat",
    "Rodent", "Gerbil", "Hamster",
  };
  // Exceptions thrown out to console:
  public static void main(String[] args)
  throws Exception {
    ArrayList pets = new ArrayList();
    try {
      Class[] petTypes = {
        Class.forName("Dog"),
        Class.forName("Pug"),
        Class.forName("Cat"),
        Class.forName("Rodent"),
        Class.forName("Gerbil"),
        Class.forName("Hamster"),
      };
      for(int i = 0; i < 15; i++)
        pets.add(
          petTypes[
            (int)(Math.random()*petTypes.length)]
            .newInstance());
    } catch(InstantiationException e) {
      System.err.println("Cannot instantiate");
      throw e;
    } catch(IllegalAccessException e) {
      System.err.println("Cannot access");
      throw e;
    } catch(ClassNotFoundException e) {
      System.err.println("Cannot find class");
      throw e;
    }
    HashMap h = new HashMap();
    for(int i = 0; i < typenames.length; i++)
```

```
      h.put(typenames[i], new Counter());
    for(int i = 0; i < pets.size(); i++) {
      Object o = pets.get(i);
      if(o instanceof Pet)
        ((Counter)h.get("Pet")).i++;
      if(o instanceof Dog)
        ((Counter)h.get("Dog")).i++;
      if(o instanceof Pug)
        ((Counter)h.get("Pug")).i++;
      if(o instanceof Cat)
        ((Counter)h.get("Cat")).i++;
      if(o instanceof Rodent)
        ((Counter)h.get("Rodent")).i++;
      if(o instanceof Gerbil)
        ((Counter)h.get("Gerbil")).i++;
      if(o instanceof Hamster)
        ((Counter)h.get("Hamster")).i++;
    }
    for(int i = 0; i < pets.size(); i++)
      System.out.println(pets.get(i).getClass());
    for(int i = 0; i < typenames.length; i++)
      System.out.println(
        typenames[i] + " quantity: " +
        ((Counter)h.get(typenames[i])).i);
  }
} ///:~
```

There's a rather narrow restriction on **instanceof**: you can compare it to a named type only, and not to a **Class** object. In the example above you might feel that it's tedious to write out all of those **instanceof** expressions, and you're right. But there is no way to cleverly automate **instanceof** by creating an **ArrayList** of **Class** objects and comparing it to those instead (stay tuned—you'll see an alternative). This isn't as great a restriction as you might think, because you'll eventually understand that your design is probably flawed if you end up writing a lot of **instanceof** expressions.

Of course this example is contrived—you'd probably put a **static** data member in each type and increment it in the constructor to keep track of the counts. You would do something like that *if* you had control of the

source code for the class and could change it. Since this is not always the case, RTTI can come in handy.

Using class literals

It's interesting to see how the **PetCount.java** example can be rewritten using class literals. The result is cleaner in many ways:

```
//: c12:PetCount2.java
// Using class literals.
import java.util.*;

public class PetCount2 {
  public static void main(String[] args)
  throws Exception {
    ArrayList pets = new ArrayList();
    Class[] petTypes = {
      // Class literals:
      Pet.class,
      Dog.class,
      Pug.class,
      Cat.class,
      Rodent.class,
      Gerbil.class,
      Hamster.class,
    };
    try {
      for(int i = 0; i < 15; i++) {
        // Offset by one to eliminate Pet.class:
        int rnd = 1 + (int)(
          Math.random() * (petTypes.length - 1));
        pets.add(
          petTypes[rnd].newInstance());
      }
    } catch(InstantiationException e) {
      System.err.println("Cannot instantiate");
      throw e;
    } catch(IllegalAccessException e) {
      System.err.println("Cannot access");
      throw e;
    }
```

```
    HashMap h = new HashMap();
    for(int i = 0; i < petTypes.length; i++)
      h.put(petTypes[i].toString(),
        new Counter());
    for(int i = 0; i < pets.size(); i++) {
      Object o = pets.get(i);
      if(o instanceof Pet)
        ((Counter)h.get("class Pet")).i++;
      if(o instanceof Dog)
        ((Counter)h.get("class Dog")).i++;
      if(o instanceof Pug)
        ((Counter)h.get("class Pug")).i++;
      if(o instanceof Cat)
        ((Counter)h.get("class Cat")).i++;
      if(o instanceof Rodent)
        ((Counter)h.get("class Rodent")).i++;
      if(o instanceof Gerbil)
        ((Counter)h.get("class Gerbil")).i++;
      if(o instanceof Hamster)
        ((Counter)h.get("class Hamster")).i++;
    }
    for(int i = 0; i < pets.size(); i++)
      System.out.println(pets.get(i).getClass());
    Iterator keys = h.keySet().iterator();
    while(keys.hasNext()) {
      String nm = (String)keys.next();
      Counter cnt = (Counter)h.get(nm);
      System.out.println(
        nm.substring(nm.lastIndexOf('.') + 1) +
        " quantity: " + cnt.i);
    }
  }
} ///:~
```

Here, the **typenames** array has been removed in favor of getting the type name strings from the **Class** object. Notice that the system can distinguish between classes and interfaces.

You can also see that the creation of **petTypes** does not need to be surrounded by a **try** block since it's evaluated at compile-time and thus won't throw any exceptions, unlike **Class.forName( )**.

When the **Pet** objects are dynamically created, you can see that the random number is restricted so it is between one and **petTypes.length** and does not include zero. That's because zero refers to **Pet.class**, and presumably a generic **Pet** object is not interesting. However, since **Pet.class** is part of **petTypes** the result is that all of the pets get counted.

A dynamic **instanceof**

The **Class isInstance** method provides a way to dynamically call the **instanceof** operator. Thus, all those tedious **instanceof** statements can be removed in the **PetCount** example:

```
//: c12:PetCount3.java
// Using isInstance().
import java.util.*;

public class PetCount3 {
  public static void main(String[] args)
  throws Exception {
    ArrayList pets = new ArrayList();
    Class[] petTypes = {
      Pet.class,
      Dog.class,
      Pug.class,
      Cat.class,
      Rodent.class,
      Gerbil.class,
      Hamster.class,
    };
    try {
      for(int i = 0; i < 15; i++) {
        // Offset by one to eliminate Pet.class:
        int rnd = 1 + (int)(
          Math.random() * (petTypes.length - 1));
        pets.add(
          petTypes[rnd].newInstance());
      }
    } catch(InstantiationException e) {
      System.err.println("Cannot instantiate");
      throw e;
```

```
    } catch(IllegalAccessException e) {
      System.err.println("Cannot access");
      throw e;
    }
    HashMap h = new HashMap();
    for(int i = 0; i < petTypes.length; i++)
      h.put(petTypes[i].toString(),
        new Counter());
    for(int i = 0; i < pets.size(); i++) {
      Object o = pets.get(i);
      // Using isInstance to eliminate individual
      // instanceof expressions:
      for (int j = 0; j < petTypes.length; ++j)
        if (petTypes[j].isInstance(o)) {
          String key = petTypes[j].toString();
          ((Counter)h.get(key)).i++;
        }
    }
    for(int i = 0; i < pets.size(); i++)
      System.out.println(pets.get(i).getClass());
    Iterator keys = h.keySet().iterator();
    while(keys.hasNext()) {
      String nm = (String)keys.next();
      Counter cnt = (Counter)h.get(nm);
      System.out.println(
        nm.substring(nm.lastIndexOf('.') + 1) +
        " quantity: " + cnt.i);
    }
  }
} ///:~
```

You can see that the **isInstance( )** method has eliminated the need for the **instanceof** expressions. In addition, this means that you can add new types of pets simply by changing the **petTypes** array; the rest of the program does not need modification (as it did when using the **instanceof** expressions).

instanceof vs. **Class** equivalence

When querying for type information, there's an important difference between either form of **instanceof** (that is, **instanceof** or

isInstance( ), which produce equivalent results) and the direct comparison of the **Class** objects. Here's an example that demonstrates the difference:

```
//: c12:FamilyVsExactType.java
// The difference between instanceof and class

class Base {}
class Derived extends Base {}

public class FamilyVsExactType {
  static void test(Object x) {
    System.out.println("Testing x of type " +
      x.getClass());
    System.out.println("x instanceof Base " +
      (x instanceof Base));
    System.out.println("x instanceof Derived " +
      (x instanceof Derived));
    System.out.println("Base.isInstance(x) " +
      Base.class.isInstance(x));
    System.out.println("Derived.isInstance(x) " +
      Derived.class.isInstance(x));
    System.out.println(
      "x.getClass() == Base.class " +
      (x.getClass() == Base.class));
    System.out.println(
      "x.getClass() == Derived.class " +
      (x.getClass() == Derived.class));
    System.out.println(
      "x.getClass().equals(Base.class)) " +
      (x.getClass().equals(Base.class)));
    System.out.println(
      "x.getClass().equals(Derived.class)) " +
      (x.getClass().equals(Derived.class)));
  }
  public static void main(String[] args) {
    test(new Base());
    test(new Derived());
  }
} ///:~
```

The **test( )** method performs type checking with its argument using both forms of **instanceof**. It then gets the **Class** reference and uses == and **equals( )** to test for equality of the **Class** objects. Here is the output:

```
Testing x of type class Base
x instanceof Base true
x instanceof Derived false
Base.isInstance(x) true
Derived.isInstance(x) false
x.getClass() == Base.class true
x.getClass() == Derived.class false
x.getClass().equals(Base.class)) true
x.getClass().equals(Derived.class)) false
Testing x of type class Derived
x instanceof Base true
x instanceof Derived true
Base.isInstance(x) true
Derived.isInstance(x) true
x.getClass() == Base.class false
x.getClass() == Derived.class true
x.getClass().equals(Base.class)) false
x.getClass().equals(Derived.class)) true
```

Reassuringly, **instanceof** and **isInstance( )** produce exactly the same results, as do **equals( )** and ==. But the tests themselves draw different conclusions. In keeping with the concept of type, **instanceof** says "are you this class, or a class derived from this class?" On the other hand, if you compare the actual **Class** objects using ==, there is no concern with inheritance—it's either the exact type or it isn't.

RTTI syntax

Java performs its RTTI using the **Class** object, even if you're doing something like a cast. The class **Class** also has a number of other ways you can use RTTI.

First, you must get a reference to the appropriate **Class** object. One way to do this, as shown in the previous example, is to use a string and the **Class.forName( )** method. This is convenient because you don't need an object of that type in order to get the **Class** reference. However, if you do

already have an object of the type you're interested in, you can fetch the **Class** reference by calling a method that's part of the **object** root class: **getClass( )**. This returns the **Class** reference representing the actual type of the object. **Class** has many interesting methods, demonstrated in the following example:

```
//: c12:ToyTest.java
// Testing class Class.

interface HasBatteries {}
interface Waterproof {}
interface ShootsThings {}
class Toy {
  // Comment out the following default
  // constructor to see
  // NoSuchMethodError from (*1*)
  Toy() {}
  Toy(int i) {}
}

class FancyToy extends Toy
    implements HasBatteries,
      Waterproof, ShootsThings {
  FancyToy() { super(1); }
}

public class ToyTest {
  public static void main(String[] args)
  throws Exception {
    Class c = null;
    try {
      c = Class.forName("FancyToy");
    } catch(ClassNotFoundException e) {
      System.err.println("Can't find FancyToy");
      throw e;
    }
    printInfo(c);
    Class[] faces = c.getInterfaces();
    for(int i = 0; i < faces.length; i++)
      printInfo(faces[i]);
    Class cy = c.getSuperclass();
```

```
    Object o = null;
    try {
      // Requires default constructor:
      o = cy.newInstance(); // (*1*)
    } catch(InstantiationException e) {
      System.err.println("Cannot instantiate");
      throw e;
    } catch(IllegalAccessException e) {
      System.err.println("Cannot access");
      throw e;
    }
    printInfo(o.getClass());
  }
  static void printInfo(Class cc) {
    System.out.println(
      "Class name: " + cc.getName() +
      " is interface? [" +
      cc.isInterface() + "]");
  }
} ///:~
```

You can see that **class FancyToy** is quite complicated, since it inherits from **Toy** and **implements** the **interface**s of **HasBatteries**, **Waterproof**, and **ShootsThings**. In **main( )**, a **Class** reference is created and initialized to the **FancyToy Class** using **forName( )** inside an appropriate **try** block.

The **Class.getInterfaces( )** method returns an array of **Class** objects representing the interfaces that are contained in the **Class** object of interest.

If you have a **Class** object you can also ask it for its direct base class using **getSuperclass( )**. This, of course, returns a **Class** reference that you can further query. This means that, at run-time, you can discover an object's entire class hierarchy.

The **newInstance( )** method of **Class** can, at first, seem like just another way to **clone( )** an object. However, you can create a new object with **newInstance( )** *without* an existing object, as seen here, because there is no **Toy** object—only **cy**, which is a reference to **y**'s **Class** object. This is a way to implement a "virtual constructor," which allows you to say "I

don't know exactly what type you are, but create yourself properly anyway." In the example above, **cy** is just a **Class** reference with no further type information known at compile-time. And when you create a new instance, you get back an **object reference**. But that reference is pointing to a **Toy** object. Of course, before you can send any messages other than those accepted by **object**, you have to investigate it a bit and do some casting. In addition, the class that's being created with **newInstance( )** must have a default constructor. In the next section, you'll see how to dynamically create objects of classes using any constructor, with the Java *reflection* API.

The final method in the listing is **printInfo( ),** which takes a **Class** reference and gets its name with **getName( ),** and finds out whether it's an interface with **isInterface( )**.

The output from this program is:

```
Class name: FancyToy is interface? [false]
Class name: HasBatteries is interface? [true]
Class name: Waterproof is interface? [true]
Class name: ShootsThings is interface? [true]
Class name: Toy is interface? [false]
```

Thus, with the **Class** object you can find out just about everything you want to know about an object.

Reflection: run-time class information

If you don't know the precise type of an object, RTTI will tell you. However, there's a limitation: the type must be known at compile-time in order for you to be able to detect it using RTTI and do something useful with the information. Put another way, the compiler must know about all the classes you're working with for RTTI.

This doesn't seem like that much of a limitation at first, but suppose you're given a reference to an object that's not in your program space. In fact, the class of the object isn't even available to your program at

compile-time. For example, suppose you get a bunch of bytes from a disk file or from a network connection and you're told that those bytes represent a class. Since the compiler can't know about the class while it's compiling the code, how can you possibly use such a class?

In a traditional programming environment this seems like a far-fetched scenario. But as we move into a larger programming world there are important cases in which this happens. The first is component-based programming, in which you build projects using *Rapid Application Development* (RAD) in an application builder tool. This is a visual approach to creating a program (which you see on the screen as a "form") by moving icons that represent components onto the form. These components are then configured by setting some of their values at program time. This design-time configuration requires that any component be instantiable, that it exposes parts of itself, and that it allows its values to be read and set. In addition, components that handle GUI events must expose information about appropriate methods so that the RAD environment can assist the programmer in overriding these event-handling methods. Reflection provides the mechanism to detect the available methods and produce the method names. Java provides a structure for component-based programming through JavaBeans (described in Chapter 13).

Another compelling motivation for discovering class information at run-time is to provide the ability to create and execute objects on remote platforms across a network. This is called *Remote Method Invocation* (RMI) and it allows a Java program to have objects distributed across many machines. This distribution can happen for a number of reasons: for example, perhaps you're doing a computation-intensive task and you want to break it up and put pieces on machines that are idle in order to speed things up. In some situations you might want to place code that handles particular types of tasks (e.g., "Business Rules" in a multitier client/server architecture) on a particular machine, so that machine becomes a common repository describing those actions and it can be easily changed to affect everyone in the system. (This is an interesting development, since the machine exists solely to make software changes easy!) Along these lines, distributed computing also supports specialized hardware that might be good at a particular task—matrix inversions, for

example—but inappropriate or too expensive for general purpose programming.

The class **Class** (described previously in this chapter) supports the concept of *reflection*, and there's an additional library, **java.lang.reflect,** with classes **Field**, **Method**, and **Constructor** (each of which implement the **Member interface**). Objects of these types are created by the JVM at run-time to represent the corresponding member in the unknown class. You can then use the **Constructor**s to create new objects, the **get( )** and **set( )** methods to read and modify the fields associated with **Field** objects, and the **invoke( )** method to call a method associated with a **Method** object. In addition, you can call the convenience methods **getFields( )**, **getMethods( )**, **getConstructors( )**, etc., to return arrays of the objects representing the fields, methods, and constructors. (You can find out more by looking up the class **Class** in your online documentation.) Thus, the class information for anonymous objects can be completely determined at run-time, and nothing need be known at compile-time.

It's important to realize that there's nothing magic about reflection. When you're using reflection to interact with an object of an unknown type, the JVM will simply look at the object and see that it belongs to a particular class (just like ordinary RTTI) but then, before it can do anything else, the **Class** object must be loaded. Thus, the **.class** file for that particular type must still be available to the JVM, either on the local machine or across the network. So the true difference between RTTI and reflection is that with RTTI, the compiler opens and examines the **.class** file at compile-time. Put another way, you can call all the methods of an object in the "normal" way. With reflection, the **.class** file is unavailable at compile-time; it is opened and examined by the run-time environment.

A class method extractor

You'll rarely need to use the reflection tools directly; they're in the language to support other Java features, such as object serialization (Chapter 11), JavaBeans (Chapter 13), and RMI (Chapter 15). However, there are times when it's quite useful to be able to dynamically extract information about a class. One extremely useful tool is a class method extractor. As mentioned before, looking at a class definition source code

or online documentation shows only the methods that are defined or overridden *within that class definition*. But there could be dozens more available to you that have come from base classes. To locate these is both tedious and time consuming[1]. Fortunately, reflection provides a way to write a simple tool that will automatically show you the entire interface. Here's the way it works:

```
//: c12:ShowMethods.java
// Using reflection to show all the methods of
// a class, even if the methods are defined in
// the base class.
import java.lang.reflect.*;

public class ShowMethods {
  static final String usage =
    "usage: \n" +
    "ShowMethods qualified.class.name\n" +
    "To show all methods in class or: \n" +
    "ShowMethods qualified.class.name word\n" +
    "To search for methods involving 'word'";
  public static void main(String[] args) {
    if(args.length < 1) {
      System.out.println(usage);
      System.exit(0);
    }
    try {
      Class c = Class.forName(args[0]);
      Method[] m = c.getMethods();
      Constructor[] ctor = c.getConstructors();
      if(args.length == 1) {
        for (int i = 0; i < m.length; i++)
          System.out.println(m[i]);
        for (int i = 0; i < ctor.length; i++)
          System.out.println(ctor[i]);
      } else {
        for (int i = 0; i < m.length; i++)
          if(m[i].toString()
```

[1] Especially in the past. However, Sun has greatly improved its HTML Java documentation so that it's easier to see base-class methods.

```
              .indexOf(args[1])!= -1)
            System.out.println(m[i]);
        for (int i = 0; i < ctor.length; i++)
          if(ctor[i].toString()
             .indexOf(args[1])!= -1)
          System.out.println(ctor[i]);
      }
    } catch(ClassNotFoundException e) {
      System.err.println("No such class: " + e);
    }
  }
} ///:~
```

The **Class** methods **getMethods( )** and **getConstructors( )** return an array of **Method** and **Constructor**, respectively. Each of these classes has further methods to dissect the names, arguments, and return values of the methods they represent. But you can also just use **toString( )**, as is done here, to produce a **String** with the entire method signature. The rest of the code is just for extracting command line information, determining if a particular signature matches with your target string (using **indexOf( )**), and printing the results.

This shows reflection in action, since the result produced by **Class.forName( )** cannot be known at compile-time, and therefore all the method signature information is being extracted at run-time. If you investigate your online documentation on reflection, you'll see that there is enough support to actually set up and make a method call on an object that's totally unknown at compile-time (there will be examples of this later in this book). Again, this is something you may never need to do yourself—the support is there for RMI and so a programming environment can manipulate JavaBeans—but it's interesting.

An enlightening experiment is to run

```
java ShowMethods ShowMethods
```

This produces a listing that includes a **public** default constructor, even though you can see from the code that no constructor was defined. The constructor you see is the one that's automatically synthesized by the compiler. If you then make **ShowMethods** a non-**public** class (that is, friendly), the synthesized default constructor no longer shows up in the

output. The synthesized default constructor is automatically given the same access as the class.

The output for **ShowMethods** is still a little tedious. For example, here's a portion of the output produced by invoking **java ShowMethods java.lang.String**:

```
public boolean
  java.lang.String.startsWith(java.lang.String,int)
public boolean
  java.lang.String.startsWith(java.lang.String)
public boolean
  java.lang.String.endsWith(java.lang.String)
```

It would be even nicer if the qualifiers like **java.lang** could be stripped off. The **StreamTokenizer** class introduced in the previous chapter can help create a tool to solve this problem:

```
//: com:bruceeckel:util:StripQualifiers.java
package com.bruceeckel.util;
import java.io.*;

public class StripQualifiers {
  private StreamTokenizer st;
  public StripQualifiers(String qualified) {
    st = new StreamTokenizer(
      new StringReader(qualified));
    st.ordinaryChar(' '); // Keep the spaces
  }
  public String getNext() {
    String s = null;
    try {
      int token = st.nextToken();
      if(token != StreamTokenizer.TT_EOF) {
        switch(st.ttype) {
          case StreamTokenizer.TT_EOL:
            s = null;
            break;
          case StreamTokenizer.TT_NUMBER:
            s = Double.toString(st.nval);
            break;
          case StreamTokenizer.TT_WORD:
```

```
            s = new String(st.sval);
            break;
          default: // single character in ttype
            s = String.valueOf((char)st.ttype);
        }
      }
    } catch(IOException e) {
      System.err.println("Error fetching token");
    }
    return s;
  }
  public static String strip(String qualified) {
    StripQualifiers sq =
      new StripQualifiers(qualified);
    String s = "", si;
    while((si = sq.getNext()) != null) {
      int lastDot = si.lastIndexOf('.');
      if(lastDot != -1)
        si = si.substring(lastDot + 1);
      s += si;
    }
    return s;
  }
} ///:~
```

To facilitate reuse, this class is placed in **com.bruceeckel.util**. As you can see, this uses the **StreamTokenizer** and **String** manipulation to do its work.

The new version of the program uses the above class to clean up the output:

```
//: c12:ShowMethodsClean.java
// ShowMethods with the qualifiers stripped
// to make the results easier to read.
import java.lang.reflect.*;
import com.bruceeckel.util.*;

public class ShowMethodsClean {
  static final String usage =
    "usage: \n" +
    "ShowMethodsClean qualified.class.name\n" +
```

```
    "To show all methods in class or: \n" +
    "ShowMethodsClean qualif.class.name word\n" +
    "To search for methods involving 'word'";
  public static void main(String[] args) {
    if(args.length < 1) {
      System.out.println(usage);
      System.exit(0);
    }
    try {
      Class c = Class.forName(args[0]);
      Method[] m = c.getMethods();
      Constructor[] ctor = c.getConstructors();
      // Convert to an array of cleaned Strings:
      String[] n =
        new String[m.length + ctor.length];
      for(int i = 0; i < m.length; i++) {
        String s = m[i].toString();
        n[i] = StripQualifiers.strip(s);
      }
      for(int i = 0; i < ctor.length; i++) {
        String s = ctor[i].toString();
        n[i + m.length] =
          StripQualifiers.strip(s);
      }
      if(args.length == 1)
        for (int i = 0; i < n.length; i++)
          System.out.println(n[i]);
      else
        for (int i = 0; i < n.length; i++)
          if(n[i].indexOf(args[1])!= -1)
            System.out.println(n[i]);
    } catch(ClassNotFoundException e) {
      System.err.println("No such class: " + e);
    }
  }
} ///:~
```

The class **ShowMethodsClean** is quite similar to the previous **ShowMethods**, except that it takes the arrays of **Method** and **Constructor** and converts them into a single array of **String**. Each of

these **String** objects is then passed through **StripQualifiers.Strip( )** to remove all the method qualification.

This tool can be a real time-saver while you're programming, when you can't remember if a class has a particular method and you don't want to go walking through the class hierarchy in the online documentation, or if you don't know whether that class can do anything with, for example, **Color** objects.

Chapter 13 contains a GUI version of this program (customized to extract information for Swing components) so you can leave it running while you're writing code, to allow quick lookups.

Summary

RTTI allows you to discover type information from an anonymous base-class reference. Thus, it's ripe for misuse by the novice since it might make sense before polymorphic method calls do. For many people coming from a procedural background, it's difficult not to organize their programs into sets of **switch** statements. They could accomplish this with RTTI and thus lose the important value of polymorphism in code development and maintenance. The intent of Java is that you use polymorphic method calls throughout your code, and you use RTTI only when you must.

However, using polymorphic method calls as they are intended requires that you have control of the base-class definition because at some point in the extension of your program you might discover that the base class doesn't include the method you need. If the base class comes from a library or is otherwise controlled by someone else, a solution to the problem is RTTI: You can inherit a new type and add your extra method. Elsewhere in the code you can detect your particular type and call that special method. This doesn't destroy the polymorphism and extensibility of the program because adding a new type will not require you to hunt for switch statements in your program. However, when you add new code in your main body that requires your new feature, you must use RTTI to detect your particular type.

Putting a feature in a base class might mean that, for the benefit of one particular class, all of the other classes derived from that base require

some meaningless stub of a method. This makes the interface less clear and annoys those who must override abstract methods when they derive from that base class. For example, consider a class hierarchy representing musical instruments. Suppose you wanted to clear the spit valves of all the appropriate instruments in your orchestra. One option is to put a **clearSpitValve( )** method in the base class **Instrument**, but this is confusing because it implies that **Percussion** and **Electronic** instruments also have spit valves. RTTI provides a much more reasonable solution in this case because you can place the method in the specific class (**Wind** in this case), where it's appropriate. However, a more appropriate solution is to put a **prepareInstrument( )** method in the base class, but you might not see this when you're first solving the problem and could mistakenly assume that you must use RTTI.

Finally, RTTI will sometimes solve efficiency problems. If your code nicely uses polymorphism, but it turns out that one of your objects reacts to this general purpose code in a horribly inefficient way, you can pick out that type using RTTI and write case-specific code to improve the efficiency. Be wary, however, of programming for efficiency too soon. It's a seductive trap. It's best to get the program working *first*, then decide if it's running fast enough, and only then should you attack efficiency issues—with a profiler.

Exercises

Solutions to selected exercises can be found in the electronic document *The Thinking in Java Annotated Solution Guide*, available for a small fee from *www.BruceEckel.com*.

1. Add **Rhomboid** to **Shapes.java**. Create a **Rhomboid**, upcast it to a **Shape**, then downcast it back to a **Rhomboid**. Try downcasting to a **Circle** and see what happens.
2. Modify Exercise 1 so that it uses **instanceof** to check the type before performing the downcast.
3. Modify **Shapes.java** so that it can "highlight" (set a flag) in all shapes of a particular type. The **toString( )** method for each derived **Shape** should indicate whether that **Shape** is "highlighted."

4. Modify **SweetShop.java** so that each type of object creation is controlled by a command-line argument. That is, if your command line is "**java SweetShop Candy**," then only the **Candy** object is created. Notice how you can control which **Class** objects are loaded via the command-line argument.

5. Add a new type of **Pet** to **PetCount3.java**. Verify that it is created and counted correctly in **main( )**.

6. Write a method that takes an object and recursively prints all the classes in that object's hierarchy.

7. Modify Exercise 6 so that it uses **Class.getDeclaredFields( )** to also display information about the fields in a class.

8. In **ToyTest.java**, comment out **Toy**'s default constructor and explain what happens.

9. Incorporate a new kind of **interface** into **ToyTest.java** and verify that it is detected and displayed properly.

10. Create a new type of container that uses a **private ArrayList** to hold the objects. Capture the type of the first object you put in it, and then allow the user to insert objects of only that type from then on.

11. Write a program to determine whether an array of **char** is a primitive type or a true object.

12. Implement **clearSpitValve( )** as described in the summary.

13. Implement the **rotate(Shape)** method described in this chapter, such that it checks to see if it is rotating a **Circle** (and, if so, doesn't perform the operation).

14. In **ToyTest.java**, use reflection to create a **Toy** object using the nondefault constructor.

15. Look up the interface for **java.lang.Class** in the HTML Java documentation from *java.sun.com*. Write a program that takes the name of a class as a command-line argument, then uses the **Class**

methods to dump all the information available for that class. Test your program with a standard library class and a class you create.

13: Programming Windows Forms

A fundamental design guideline is "make simple things easy, and difficult things possible."[1]

There have been two antagonistic forces at work in the Windows GUI programming world – the ease of use epitomized by Visual Basic and the control available to C programmers using the Win32 API. Reconciling these forces is one of the great achievements of the .NET Framework. Visual Studio .NET provides a VB-like toolset for drawing forms, dragging-and-dropping widgets onto them, and rapidly specifying their properties, event, and responses. However, this easy-to-use tool sits on top of Windows Forms and GDI+, systems whose object-oriented design, consistency, and flexibility are unsurpassed for creating rich client interfaces.

The aim of this chapter is to give a good understanding of the underlying concepts at play in implementing graphical user interfaces and to couple those concepts with concrete details on the most commonly-used widgets.

For Java programmers, the ease with which rich, highly responsive interfaces and custom components can be created in C# will come as a revelation. Java's UI models have been based on the premise that operating system integration is superfluous to the task of creating

---

[1] A variation on this is called "the principle of least astonishment," which essentially says: "don't surprise the user."

program interfaces. This is absurd, if for no other reason than its violation of

Delegates

One of C#'s most interesting language features is its support for *delegates*, an object-oriented, method type. The line:

```
delegate string Foo(int param);
```

is a *type declaration* just as is:

```
class Bar{ }
```

And just as to be useful a class type has to be instantiated (unless it just has **static** data and methods), so too must delegate types be instantiated to be of any use. A delegate can be instantiated with any method that matches the delegate's type signature. Once instantiated, the delegate reference can be used directly as a method. Delegates are object-oriented in that they can be bound not just to static methods, but to instance methods; in doing so, a delegate will execute the specified method on the designated object. A simple example will show the basic features of delegates:

```
//:c13:Profession.cs
//Declaration and instantiation of delegates

delegate void Profession();

class ProfessionSpeaker{

   static void StaticSpeaker(){
      System.Console.WriteLine("Medicine");
   }

   static int doctorIdCounter = 0;
   int doctorId = doctorIdCounter++;

   void InstanceSpeaker(){
      System.Console.WriteLine("Doctor " + doctorId);
   }
```

```
    int DifferentSignature(){
       System.Console.WriteLine("Firefighter");
       return 0;
    }

    public static void Main(){
       //declare delegate reference (== null)
       Profession p;
       //instantiate delegate reference
       p = new Profession(
          ProfessionSpeaker.StaticSpeaker);
       p();

       ProfessionSpeaker s1 = new ProfessionSpeaker();
       ProfessionSpeaker s2 = new ProfessionSpeaker();

       //"instantiate" to specific instances
       Profession p1 = new Profession(
          s1.InstanceSpeaker);
       Profession p2 = new Profession(
          s2.InstanceSpeaker);


       p1();
       p2();

       //Won't compile, different signature
       //Profession p3 = new Profession(
       //   s2.DifferentSignature);
     }
}///:~
```

The **Profession** delegate type is declared to take no parameters and to return **void**. The **ProfessionSpeaker** has two methods with this signature: a **static StaticSpeaker()** method and an instance method called **InstanceSpeaker()**. **ProfessionSpeaker** has a static **doctorIdCounter** which is incremented every time a new **ProfessionSpeaker** is instantiated; thus **InstanceSpeaker()** has different output for each **ProfessionSpeaker** instance.

ProfessionSpeaker.Main() declares a delegate reference **Profession p**. Like all declared but not initialized variables, **p** is null at this point. The next line is the delegate's "constructor" and is of the form:

```
new DelegateTypeName( NameOfMethodToDelegate );
```

This first delegate is instantiated with the **StaticSpeaker()** method (note that the value passed to the delegate "constructor" is just the *name* of the method without the parentheses that would specify an actual method *call*).

Once the delegate is initialized, the reference *variable* acts as if it was an in-scope method *name*. So the line **p()** results in a call to **ProfessionSpeaker.StaticSpeaker()**.

Main() then instantiates two new **ProfessionSpeaker**s; one of which will have a **doctorId** of 0 and the other of 1. We declare two new **Profession** delegates, but this time use a slight different form:

```
new DelegateTypeName( objectReference.MethodName );
```

You can think of this as passing "the instance of the method" associated with the **objectReference**, even though "that's not how it really is" in memory (there's a copy of each piece of *data* for an instance, but *methods* are not duplicated). The two delegates **p1** and **p2** can now be used as proxies for the methods **s1.InstanceSpeaker()** and **s2.InstanceSpeaker()**.

The delegated-to method must have the exact parameters *and return type* of the delegate type declaration. Thus, we can't use **DifferentSignature()** (which returns an **int**) to instantiate a **Profession** delegate.

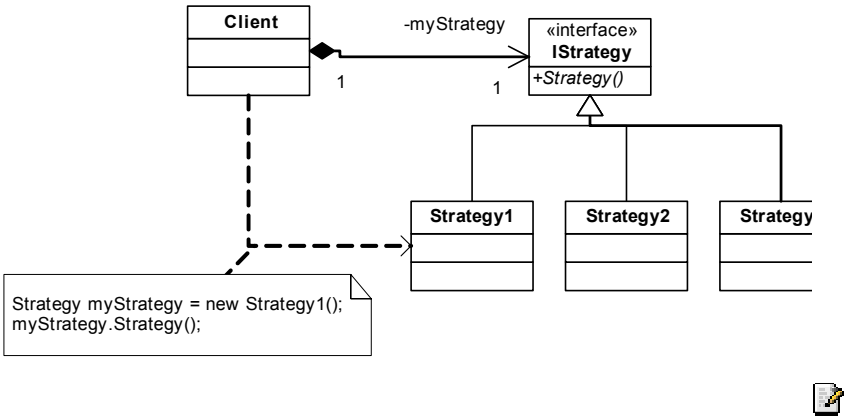
When run, the calls to the delegates **p, p1**, and **p2** result in this output:

```
Medicine
Doctor 0
Doctor 1
```

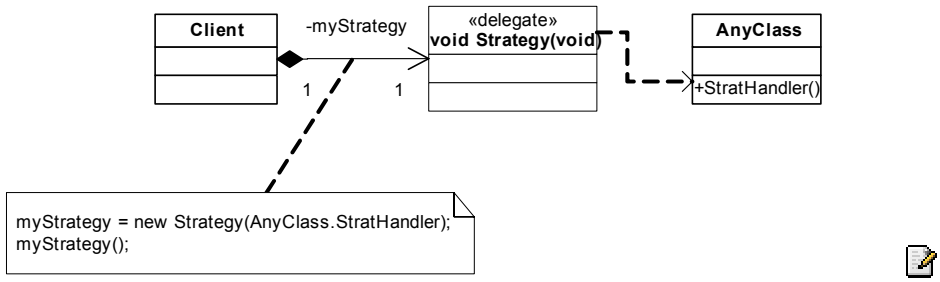
Designing With Delegates

Delegates are used extensively in Windows Forms, .NET's object-oriented framework for GUI programming. However, they can be effectively used

in any situation where variables in behavior are more important than variables in state. We've already talked about the Strategy pattern (#ref#):



Delegates provide an alternate design:



The Strategy pattern explicitly binds the implementation to its use satisfying the contract of the interface **IStrategy**; any class which will be used as a **Strategy** must declare itself as **: IStrategy** and implement the method **void Strategy()**. With delegates, there is no explicit binding of the handler to the delegate type; a new programmer coming to the situation wouldn't necessarily know that **AnyClass.StratHandler()** was being used to instantiate a **Strategy** delegate. This might lead to trouble: the new programmer could change the behavior of **StratHandler()** in a way inappropriate to its use as a **Strategy** delegate

and the change would not be caught by either the compiler or by **AnyClass**'s unit tests; the problem wouldn't appear until **Client**'s unit tests were run.

On the other hand, the delegate model is significantly less typing and can delegate handling to a method in a type for which you don't have source code (although there's no obvious scenario that would call for that).

MORE HERE

Multicast Delegates

Delegates that return type void may designate a *series* of methods that will be invoked when the delegate is called. Such delegates are called *multicast delegates*. Methods can be added and remove from the call chain by using the overloaded += and -= operators. Here's an example:

```
//:c13:Multicast.cs
//Demonstrates multicast delegates

class Rooster{
   internal void Crow(){
      System.Console.WriteLine("Cock-a-doodle-doo");
   }

}

class PaperBoy{
   internal void DeliverPapers(){
      System.Console.WriteLine("Throw paper on roof");
   }
}

class Sun{
   internal void Rise(){
      System.Console.WriteLine("Spread rosy fingers");
   }
}

class DawnRoutine{
```

```
   delegate void DawnBehavior();
   DawnBehavior multicast;

   DawnRoutine(){
      multicast = new DawnBehavior(
         new Rooster().Crow);
      multicast += new DawnBehavior(
         new PaperBoy().DeliverPapers);
      multicast += new DawnBehavior(
         new Sun().Rise);
   }

   void Break(){
      multicast();
   }

   public static void Main(){
      DawnRoutine dr = new DawnRoutine();
      dr.Break();
   }
}///:~
```

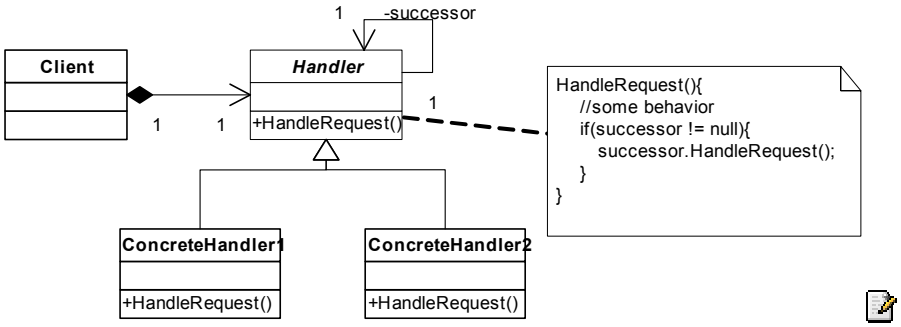
After declaring three classes (**Rooster**, **PaperBoy**, and **Sun**) that have methods associated with the dawn, we declare a **DawnRoutine** class and, within its scope, declare a **DawnBehavior** delegate type and an instance variable **multicast** to hold the instance of the delegate. The **DawnRoutine()** constructor's first line instantiates the delegate to the **Crow()** method of a **Rooster** object (the garbage collector is smart enough to know that although we're using an *anonymous* instance of **Rooster**, the instance will not be collected as long as the delegate continues to hold a reference to it).

New instances of **DawnBehavior** are instantiated with references to "instances" of **PaperBoy.DeliverPapers()** and **Sun.Rise()**. These DawnBehavior delegates are added to the **multicast** delegate with the += operator. **Break()** invokes **multicast()** once, but that single call in turn is *multicast* to all the delegates:

```
Cock-a-doodle-doo
Throw paper on roof
Spread rosy fingers
```

Multicast delegates are used throughout Windows Forms to create event-handling chains, each of which is responsible for a particular aspect of the total desired behavior (perhaps display-oriented, perhaps logical, perhaps something that writes to a logfile).

Multicast delegates are similar to the *Chain of Responsibility* design pattern:



The one difference is that the *Chain of Responsibility* is usually implemented so that an individual handler can decide to end the processing chain (that is, a handler may decide not to call it's **successor**'s **HandleRequest()** method). It is not possible for a delegated-to method to have this sometimes-desired option, so there may be times when you implement a *Chain of Responsibility* within an interface on which you create single or multicast delegates:

```
//:c13:DelegatedChainOfResponsibility.cs
//Shows polymorphic delegation, plus CoR pattern
using System;

interface Handler{
   void HandleRequest();
}

class ConcreteHandler1 : Handler{
   Random r = new Random();
   Handler successor = new ConcreteHandler2();

   public void HandleRequest() {
```

```
      if(r.NextDouble() > 0.5){
         System.Console.WriteLine(
            "CH1: Handling incomplete");
         successor.HandleRequest();
      }else{
         System.Console.WriteLine(
            "CH1: Handling complete");
      }
   }
}

class ConcreteHandler2 : Handler{
   public void HandleRequest() {
      System.Console.WriteLine("H2 handling");
   }
}

class Client{
   delegate void PolyDel();

  private static void AlsoHandler(){
      System.Console.WriteLine("AlsoHandler");
   }

   public static void Main(){
      //Note upcast
      Handler h = new ConcreteHandler1();
      PolyDel del = new PolyDel(h.HandleRequest);
      del += new PolyDel(AlsoHandler);

      del();
   }
}
```

The **Handler** interface and its two subtypes implement the *Chain of Responsibility* pattern. **ConcreteHandler1** "flips a coin" in the form of a random number and either calls **successor.HandleRequest()** or not. The **PolyDel** delegate declared within **Client** matches the interface in **Handler**, so when a new **Handler** is instantiated in **Client.Main()**, there is no problem in using that handle's **HandleRequest** method as the argument to a new **PolyDel** delegate. Since delegates are object-

oriented, there is no problem using a handle that's been upcast to an abstract data type in the construction of a delegate. When **del()** is invoked, half the time the **ConcreteHandler1** will forward a call to its **ConcreteHandler2 successor** and half the time not. Because **PolyDel** is a multicast handler, though, **AlsoHandler()** will always be called.

Events

The behavior of a multicast delegate can be interpreted as a single event (the multicast method call) causing multiple behaviors (the calls on the various delegated-to methods). However, with a normal delegate, the delegated-to method has no way of knowing anything about the context in which it was invoked. In our **Multicast** example, the **Sun.Rise()** method (say) could not have affected any change in the **DawnRoutine** object that invoked it. By convention, multicast delegates that require context should be of type:

```
delegate void DelegateName(
      object source, EventArgsSubtype x);
```

The **EventArgs** class is defined in the **System** namespace and, by default, contains nothing but a static readonly property **Empty** that is defined as returning an **EventArg** equivalent to an **EventArg** created with a no-args constructor (in other words, a generic, undistinguishable, and therefore "Empty" argument).

Subtypes of **EventArgs** are expected to define and expose properties that are the most likely important pieces of context. For instance, the **DawnBehavior** delegate might be paired with a **DawnEventArgs** object that contained the weather and time of sunrise.

If a class wishes to define a multicast delegate of this sort and expose it publicly as a property, the normal C# syntax would be:

```
void DawnDelegate(object source, DawnEventArgs dea);
class DawnEventArgs : EventArgs{}

class DawnBehavior{
   private DawnDelegate d;
   public DawnDelegate DawnEvent{
      get { return d; }
```

```
      set { d = value; }
   }
}
```

A shortcut is to simply declare an *event property*:

```
public event DawnDelegate DawnEvent;
```

You still have to define the delegate and the subtype of **DawnEventArgs** and there is no difference in behavior between a public multicast delegate exposed as a normal property and one exposed as an event property. However, event properties may be treated differently by developer tools such as the doc-comment generator or visual builder tools such as the one in Visual Studio.NET.

This example shows event handlers that modify their behavior depending on the context:

```
//:c13:EventProperty.cs
using System;

delegate void DawnDelegate(
      object source, DawnEventArgs dea);

enum Weather{ sunny, rainy };

class DawnEventArgs : EventArgs{
   //Hide base class constructor
   private DawnEventArgs() : base(){}
   public DawnEventArgs(Weather w){
      this.w = w;
   }

   private Weather w;
   public Weather MorningWeather{
      set { w = value; }
      get { return w; }
   }
}

class DawnBehavior{
   public event DawnDelegate DawnEvent;
```

```
   public static void Main(){
      DawnBehavior db = new DawnBehavior();
      db.DawnEvent = new DawnDelegate(
         new Rooster().Crow);
      db.DawnEvent += new DawnDelegate(
         new Sun().Rise);
      DawnEventArgs dea =
         new DawnEventArgs(Weather.sunny);
      db.DawnEvent(typeof(DawnBehavior), dea);
      dea =
         new DawnEventArgs(Weather.rainy);
      db.DawnEvent(typeof(DawnBehavior), dea);
   }
}

class Rooster{
   internal void Crow(object src, DawnEventArgs dea){
      if(dea.MorningWeather == Weather.sunny){
         System.Console.WriteLine(
            "Cock-a-doodle-doo");
      }else{
         System.Console.WriteLine("Sleep in");
      }
   }
}

class Sun{
   internal void Rise(object src, DawnEventArgs dea){
      if(dea.MorningWeather == Weather.sunny){
         System.Console.WriteLine(
            "Spread rosy fingers");
      }else{
         System.Console.WriteLine(
            "Cast a grey pall");
      }
   }
}
```

In this example, the **DawnEvent** is created in the static **Main()** method, so we couldn't send **this** as the source nor does passing in the **db** instance seem appropriate. We could pass **null** as the source, but passing null is generally a bad idea. Since the event is created by a static method and a

static method is associated with the class, it seems reasonable to say that the source is the type information of the class, which is retrieved by **typeof(DawnBehavior)**.

Recursive Traps

Conceptually, event-driven programs are asynchronous – when an event is "fired" (or "raised" or "sent"), control returns to the firing method and, sometime in the future, the event handler gets called. In reality, C#'s events are synchronous, meaning that control does not return to the firing method until the event handler has completed. This conceptual gap can lead to serious problems. If the event handler of event *X* itself raises events, and the handling of these events results in a new event *X*, the system will recurse, eventually either causing a stack overflow exception or exhausting some non-memory resource.

In this example, a utility company sends bills out, a homeowner pays them, which triggers a new bill. From a conceptual standpoint, this should be fine, because the payment and the new bill are separate events.

```
//:c13:RecursiveEvents.cs
//Demonstrates danger in C# event model
using System;

delegate void PaymentEvent(object src, BillArgs ea);

class BillArgs{
   internal BillArgs(double c){
      cost = c;
   }
   public double cost;
}

abstract class Bookkeeper{
   public event PaymentEvent Inbox;

   public static void Main(){
      Bookkeeper ho = new Homeowner();
      UtilityCo uc = new UtilityCo();

      uc.BeginBilling(ho);
```

```
    }

    internal void Post(object src, double c){
       Inbox(src, new BillArgs(c));
    }
}

class UtilityCo : Bookkeeper{
    internal UtilityCo(){
       Inbox += new PaymentEvent(this.ReceivePmt);
    }

    internal void BeginBilling(Bookkeeper bk){
       bk.Post(this, 4.0);
    }

    public void ReceivePmt(object src, BillArgs ea){
       Bookkeeper sender = src as Bookkeeper;
       System.Console.WriteLine("Received pmt from " +
          sender);
       sender.Post(this, 10.0);
    }
}


class Homeowner : Bookkeeper{
    internal Homeowner(){
       Inbox += new PaymentEvent(ReceiveBill);
    }
    public void ReceiveBill(object src, BillArgs ea){
       Bookkeeper sender = src as Bookkeeper;
       System.Console.WriteLine("Writing check to " +
          sender + " for " + ea.cost);
       sender.Post(this, ea.cost);
    }
}
```

First, we declare a delegate type called **PaymentEvent** which takes as an argument a **BillArgs** reference containing the amount of the bill or payment. We then create an abstract **Bookkeeper** class with a **PaymentEvent** event called **Inbox**. The **Main()** for the sample creates

a **HomeOwner**, a **UtilityCo,** and passes the reference to the **HomeOwner** to the **UtilityCo** to begin billing. **Bookkeeper** then defines a method called **Post()** which triggers the **PaymentEvent( );** we'll explain the rationale for this method in a little bit.

UtilityCo.BeginBilling() takes a **Bookkeeper** (the homeowner) as an argument. It calls that **Bookkeeper**'s **Post( )** method, which in turn will call that **Bookkeeper**'s **Inbox** delegate. In the case of the **Homeowner**, that will activate the **ReceiveBill( )** method. The homeowner "writes a check" and **Post( )**s it to the source. If events were asynchronous, this would not be a problem.

However, when run, this will run as expected for several hundred iterations, but then will crash with a stack overflow exception. Neither of the event handlers (**ReceiveBill( )** and **ReceivePayment( )**) ever returns, they just recursively call each other in what would be an infinite loop but for the finite stack. More subtle recursive loops are a challenge when writing event-driven code in C#.

Perhaps in order to discourage just these types of problems, event properties differ from delegates in one very important way: an event can only be invoked by the very class in which it is declared; even descendant types cannot directly invoke an event. This is why we needed to write the **Post( )** method in **Bookkeeper**, **HomeOwner** and **UtilityCo** cannot execute **Inbox( )**, attempting to do so results in a compilation error.

This language restriction is a syntactical way of saying "raising an event is a big deal and must be done with care." Event-driven designs may require multiple threads in order to avoid recursive loops (more on this in chapter #threading#). Or they may not. This restriction on events does not force you into any particular design decisions – as we showed in this example, one can simply create a public proxy method to invoke the event.

The Genesis of Windows Forms

While C# events are not asynchronous, "real" Windows events are. Behind the scenes, Windows programs have an event loop that receives

unsigned integers corresponding to such things as mouse movements and clicks and keypresses, and then say "If that number is *x*, call function *y*." This was state-of-the-art stuff in the mid-1980s before object-orientation became popular. In the early 1990s, products such as Actor, SQL Windows, Visual Basic, and MFC began hiding this complexity behind a variety of different models, often trading off object-oriented "purity" for ease of development or performance of the resulting application.

Although programming libraries from companies other than Microsoft were sometimes superior, Microsoft's libraries always had the edge in showcasing Windows latest capabilities. Microsoft parlayed that into increasing market share in the development tools category, at least until the explosion of the World Wide Web in the mid-1990s, when the then-current wisdom about user interfaces (summary: "UIs must consistently follow platform standards, and UIs must be fast") was left by the wayside for the new imperative ("All applications must run inside browsers").

One of the programming tools that had difficulty gaining marketshare against Microsoft was Borland's Delphi, which combined a syntax derived from Turbo Pascal, a graphical builder a la Visual Basic, and an object-oriented framework for building UIs. Delphi was the brainchild of Anders Hejlsberg, who subsequently left Borland for Microsoft, where he developed the predecessor of .NET's Windows Forms library for Visual J++. Hejlsberg was the chief designer of the C# language and C#'s delegates trace their ancestry to Delphi. (Incidentally, Delphi remains a great product and is now, ironically, the best tool for programming native Linux applications!)

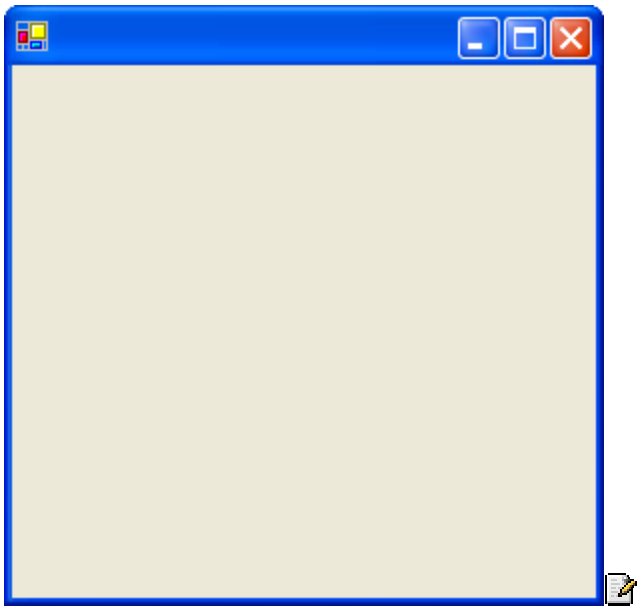
So Windows Forms is an object-oriented wrapper of the underlying Windows application. The doubling and redoubling of processor speed throughout the 1990s has made any performance hit associated with this type of abstraction irrelevant; Windows Forms applications translate the raw Windows events into calls to multicast delegates (i.e., **event**s) so efficiently that most programmers will never have a need to side-step the library.

Creating a Form

With Windows Forms, the static Main() method calls the static method Application.Run(), passing to it a reference to a subtype of **Form**. All the behavior associated with creating, displaying, closing and otherwise manipulating a Window (including repainting, a finicky point of the "raw" Windows API) is in the base type **Form** and need not be of concern to the programmer. Here's a fully functional Windows Form program:

```
//:c13:FirstForm.cs
using System.Windows.Forms;

class FirstForm : Form{
   public static void Main(){
      Application.Run(new FirstForm());
   }
}///:~
```

that when run produces this window:



The base class **Form** contains more than 100 public and protected properties, a similar number of methods, and more than 70 events and corresponding event-raising methods. But it doesn't stop there; **Form** is a

subtype of a class called **Control** (not a direct subtype, it's actually **Form** : **ContainerControl** : **ScrollableControl** : **Control**). Instances of **Control** have a property called **Controls** which contains a collection of other controls. This structure, an example of the *Composite* design pattern, allows everything from simple buttons to the most complex user-interfaces to be treated uniformly by programmers and development tools such as the visual builder tool in Visual Studio .NET.

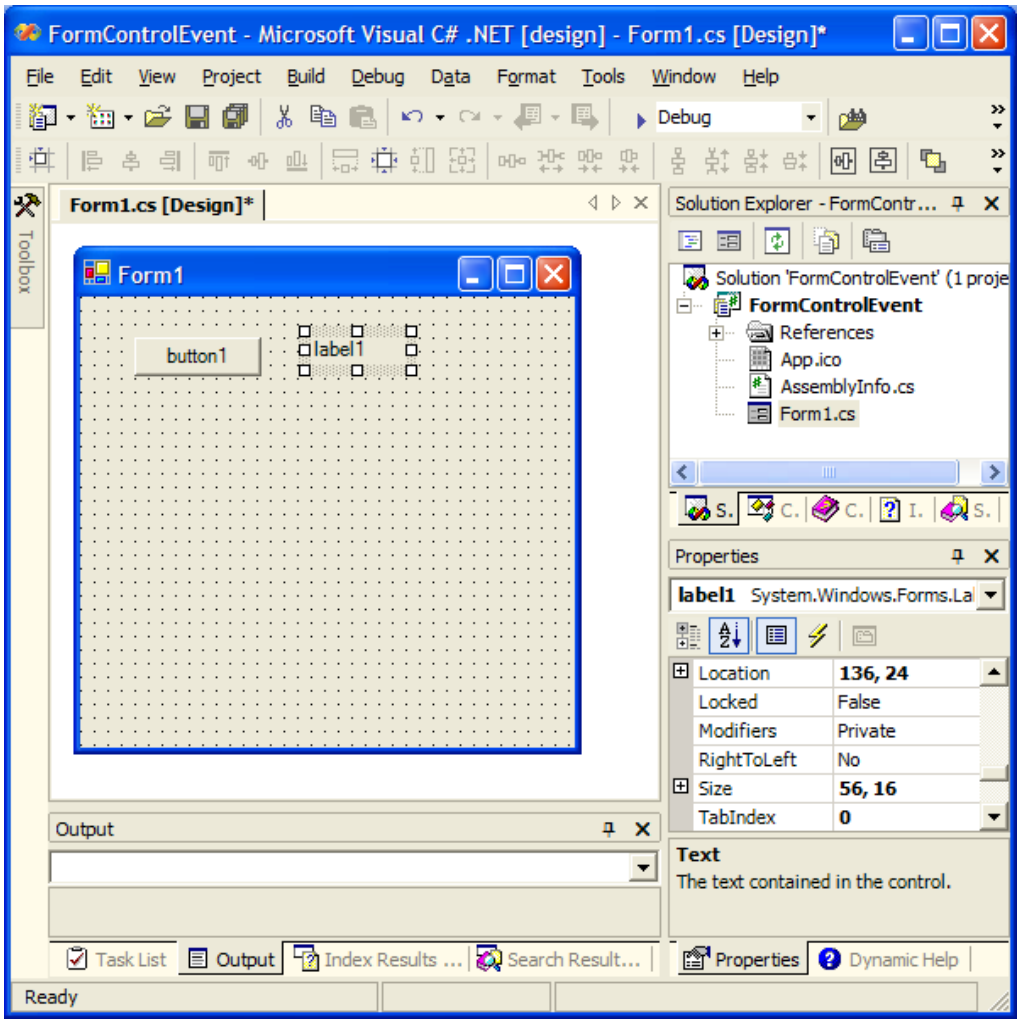
GUI Architectures

Architectures were presented in chapter #architectur# as an "overall ordering principle" of a system or subsystem. While the **Controls** property of **Control** is an ordering principle for the static structure of the widgets in a Windows Forms application, Windows Forms does not dictate an ordering principle for associating these widgets with particular events and program logic. Several architectures are possible with Windows Forms, and each has its strengths and weaknesses.

It's important to have a consistent UI architecture because, as Larry Constantine and Lucy Lockwood point out, while the UI is just one, perhaps uninteresting, part of the system to the programmer, to the end user, the UI *is* the program. The UI is the entry point for the vast majority of change requests, so you'd better make it easy to change the UI without changing the logical behavior of the program. Decoupling the presentation layer from the business layer is a fundamental part of professional development.

Using the Visual Designer

Open Visual Studio .NET, bring up the New Project wizard, and create a Windows Application called FormControlEvent. The wizard will generate some source code and present a "Design View" presentation of a blank form. Drag and drop a button and label onto the form. You should see something like this:



In the designer, double-click the button. Visual Studio will switch to a code-editing view, with the cursor inside a method called **button1_Click()**. Add the line;

```
label1.Text = "Clicked";
```

The resulting program should look a lot like this:

```
//:c13:FormControlEvent.cs
//Designer-generated Form-Control-Event architecture
using System;
```

```
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace FormControlEvent{
   /// <summary>
   /// Summary description for Form1.
   /// </summary>
   public class Form1 : System.Windows.Forms.Form {
      private System.Windows.Forms.Label label1;
      private System.Windows.Forms.Button button1;
      /// <summary>
      /// Required designer variable.
      /// </summary>
      private System.ComponentModel.Container
      components = null;

      public Form1(){
         //
         // Required for Windows Form Designer support
         //
         InitializeComponent();

         //
         // TODO: Add any constructor code after
         // InitializeComponent call
         //
      }

      /// <summary>
      /// Clean up any resources being used.
      /// </summary>
      protected override void Dispose(
          bool disposing ){
         if ( disposing ) {
            if (components != null) {
               components.Dispose();
            }
         }
```

```
            base.Dispose( disposing );
        }

#region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support
        /// - do not modify the contents of this method
        /// with the code editor.
        /// </summary>
        private void InitializeComponent(){
            this.label1 =
                new System.Windows.Forms.Label();
            this.button1 =
                new System.Windows.Forms.Button();
            this.SuspendLayout();
            //
            // label1
            //
            this.label1.Location =
                new System.Drawing.Point(136, 24);
            this.label1.Name = "label1";
            this.label1.Size =
                new System.Drawing.Size(56, 16);
            this.label1.TabIndex = 0;
            this.label1.Text = "label1";
            //
            // button1
            //
            this.button1.Location =
                new System.Drawing.Point(32, 24);
            this.button1.Name = "button1";
            this.button1.TabIndex = 1;
            this.button1.Text = "button1";
            this.button1.Click +=
             new System.EventHandler(this.button1_Click);
            //
            // Form1
            //
            this.AutoScaleBaseSize =
                new System.Drawing.Size(5, 13);
            this.ClientSize =
```

```
            new System.Drawing.Size(292, 266);
         this.Controls.AddRange(
            new System.Windows.Forms.Control[]{
                                 this.button1,
                                 this.label1});
         this.Name = "Form1";
         this.Text = "Form1";
         this.ResumeLayout(false);

      }
#endregion

      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main() {
         Application.Run(new Form1());
      }

      private void button1_Click(
         object sender, System.EventArgs e) {
         label1.Text = "Clicked";
      }
   }
}
```

The first interesting detail is the **#region** - **#endregion** statements. These preprocessing directives (see Chapter #preprocessing#) delineate a code section that Visual Studio .NET may collapse in its "outlining" mode; indeed, when you first switch to code view, this area of the code was probably somewhat hidden from view. While it's generally a good idea to heed the warning about not editing Designer-generated code, the code is well worth taking a closer look at.

The label and button that we dragged onto the form are initialized as new objects from the System.Windows.Forms namespace. The call to **SuspendLayout()** indicates that a series of manipulations are coming and that each individual one should not trigger the potentially expensive layout calculation on the **Control** and all of its sub-**Controls**. Some of the basic properties for each control are then set:

- **Location** specifies the point where the upper-left corner of the control is relative to the upper-left corner of the containing control or, if the **Control** is a **Form Location** is the screen coordinates of the upper-left corner (including the **Form**'s border if it has one, as most do). This is a value that you can freely manipulate without worrying about the

- **Size** is measured in pixels. Like **Location**, this property returns a value, not a reference, so to manipulate the **Control**, you must assign any change to the property to have any affect:

```
Size s = myControl.Size;
s.Width += 10; //not a reference, no change to control
myControl.Size = s; //Now control will change
```

- **TabIndex** specifies the order in which a control is activated when the user presses the Tab key.

- **Text** is displayed in various ways, depending upon the **Control**'s type. The **Text** of the form, for instance, is displayed as the Window's title, while the **Button** and **Label** have other properties such as **TextAlign** and **Font** to fine-tune their appearance (**Form** has a **Font** property, too, but it just sets the default font for its sub-controls; it does not change the way the title of the **Form** is displayed). The **Name** property corresponds to the named variable that represents the control and is necessary for the visual designer to work; don't manually change this.

  The final part of the block of code associated with **button1** reads:

```
this.button1.Click +=
   new System.EventHandler(this.button1_Click);
```

  From our previous discussion of multicast delegates, this should be fairly easy to interpret: **Button** has an event property **Click** which specifies a multicast delegate of type **EventHandler**. The method **this.button1_Click()** is being added as a multicast listener.

  At the bottom of the **InitializeComponent** method, additional properties are set for the **Form1** itself. **AutoScaleBaseSize** specifies how the **Form** will react if the **Form**'s font is changed to a different size (as can happen by default in Windows). **ClientSize** is the area of the **Control** in which other **Control**'s can be placed; in the case of a window,

that excludes the title bar and border, scrollbars are also not part of the *client area*.

The method **Controls.AddRange()** places an array of **Control**s in the containing **Control**. There is also an **Add()** method which takes a single control, but the visual designer always uses **AddRange()**.

Finally, **ResumeLayout(),** the complement to **SuspendLayout()**, reactivates layout behavior. The visual designer passes a **false** parameter, indicating that it's not necessary to force an immediate re-layout of the **Control**.

The **Main()** method is prepended with an **[STAThread]** attribute, which sets the threading model to "single-threaded apartment." We'll discuss threading models in chapter #threading#.

The last method is the **private** method **button1_Click()**, which was attached to **button1**'s **Click** event property in the **InitializeComponent()** method. In this method we directly manipulate the **Text** property of the **label1** control.

Some obvious observations about the output of the visual designer: it works with code that is both readable and (despite the warning) editable, the visual designer works within the monolithic **InitializeComponent()** except that it creates event-handler methods that are in the same **Control** class being defined, and the code isn't "tricky" other than the **[STAThread]** attribute and the **Dispose()** method (a method which is not necessary unless the **Control** or one of its sub-controls contains non-managed resources, as discussed in chapter #dispose#).

Less obviously, taken together, the visual designer *does* implicitly impose "an overall ordering principle" to the system. The visual designer constructs applications that have a statically structured GUI, individual identity for controls and handlers, and localized event-handling.

The problem with this architecture, as experienced Visual Basic programmers can attest, is that people can be fooled into thinking that the designer is "doing the object orientation" for them and event-handling routines become monolithic procedural code chunks. This can also lead to

people placing the domain logic directly in handlers, thus foregoing the whole concept of decoupling UI logic from domain logic.

This is a prime example of where sample code such as is shown in articles or this book is misleading. Authors and teachers will generally place domain logic inline with a control event in order to save space and simplify the explanation (as we did with the **label1.Text = "Clicked"** code). However, in professional development, the structure of pretty much any designer-generated event handler should probably be:

```
private void someControl_Click(
    object sender, System.EventArgs e) {
       someDomainObject.SomeLogicalEvent();
}
```

This structure separates the concept of the **Control** and GUI events from domain objects and logical events and a GUI that uses this structure will be able to change its domain logic without worrying about the display details.

Unfortunately, alerting domain objects to GUI events is only half the battle, the GUI must somehow reflect changes in the state of domain objects. This challenge has several different solutions.

Form-Event-Control

The first GUI architecture we'll discuss could be called "Form-Event-Control." The FEC architecture uses a unified event-driven model: GUI objects create GUI events that trigger domain logic that create domain events that trigger GUI logic. This is done by creating domain event properties and having controls subscribe to them, as this example shows:

```
//:c13:FECDomain.cs
using System.Text.RegularExpressions;

delegate void StringSplitHandler(
   object src, SplitStringArgs args);

class SplitStringArgs : EventArgs {
   private SplitStringArgs(){}
```

```
   public SplitStringArgs(string[] strings){
      this.strings = strings;
   }

   string[] strings;
   public string[] Strings{
      get { return strings;}
      set { strings = value;}
   }
}

class DomainSplitter {
   Regex re = new Regex("\\s");
   string[] substrings;

   public event StringSplitHandler StringsSplit;

   public void SplitString(string inStr){
      substrings = re.Split(inStr);
      StringsSplit(
         this, new SplitStringArgs(substrings));
   }
}///:~
```

This is our domain object, which splits a string into its substrings with the **Regex.Split()** method. When this happens, the **DomainSplitter** raises a **StringsSplit** event with the newly created substrings as an argument to its **SplitStringArgs**. Now to create a Windows Form that interacts with this domain object:

```
//:c13:FECDomain2.cs
//Compile with csc FECDomain FECDomain2
using System;
using System.Drawing;
using System.Windows.Forms;

class FECDomain : Form {
   TextBox tb = new TextBox();
   Button b = new Button();
   Label[] labels;
   DomainSplitter domainObject = new DomainSplitter();
```

```
    FECDomain(){
       tb.Location = new Point(10, 10);
       tb.Text = "The quick brown fox";
       b.Location = new Point(150, 10);
       b.Text = "Split text";

       b.Click += new EventHandler(this.GUIEvent);
       domainObject.StringsSplit +=
         new StringSplitHandler(this.DomainEvent);

       this.Text = "Form-Event-Control";
       this.Controls.Add(tb);
       this.Controls.Add(b);
    }

    void GUIEvent(object src, EventArgs args){
       domainObject.SplitString(tb.Text);
    }

    void DomainEvent(object src, SplitStringArgs args){
       string[] strings = args.Strings;
       if (labels != null) {
          foreach(Label l in labels){
             this.Controls.Remove(l);
          }
       }
       labels = new Label[strings.Length];
       int row = 40;
       for (int i = 0; i < labels.Length; i++) {
          labels[i] = new Label();
          labels[i].Text = strings[i];
          labels[i].Location = new Point(100, row);
          row += 20;
       }
       this.Controls.AddRange(labels);
    }

    public static void Main(){
       Application.Run(new FECDomain());
    }
}///:~
```

Obviously, we didn't use Visual Studio's designer to build this form but have reverted to working directly from within a code editor. Our **FECDomain** form contains a text box, a button, an array of Label controls, and a reference to **DomainSplitter**.

The first part of the **FEDomain()** constructor specifies the location and text of the text box and button. We then specify two delegates: **GUIEvent** is a delegate of type **EventHandler** and is attached to the button's **Click** event property and **DomainEvent** is of type **StringSplitHandler** and is attached to the **DomainSplitter**'s **StringSplit** event. The final part of the constructor adds the textbox and button to the form.

When the button is pressed, the **Click** delegate invokes the **GUIEvent()** method, which passes the text of the textbox to the **domainObject.SplitString()** logical event. This in turn will raise a **StringSplit** event that calls back to the **DomainEvent()** method.

The **DomainEvent()** method creates and displays a label for each of the individual strings. The first time **DomainEvent()** is called, the **labels** array will be null because we do not initialize it in the constructor. If, though, **labels** is not null, we remove the existing labels from the **Controls** collection. We initialize the **labels** array to be able to hold a sufficient number of references and then initialize individual labels with the appropriate **string** and a new position. Once all the **labels** are created, **Controls.AddRange()** adds them to the **FECDomain**'s client area.

The FEC architecture is vulnerable to the recursive loops problems discussed previously. If a domain event triggers a GUI handler which in turn activates the relevant domain event, the system will recurse and crash (when dealing with GUIs, the crash exception typically involves starvation of some Windows resource before the stack overflows). However, FEC is very straightforward – although in the tiny programs that illustrate a book it is more complex than just putting domain logic directly into a GUI event handler, in practice it will very likely be less complex and provides for a very clean and understandable separation of GUI and domain logic.

Presentation-Abstraction-Control

An alternative GUI architecture to FEC proposes that the whole concept of separating domain logic from **Control**s is overemphasized. In this view, flexibility is achieved by encapsulating all the display, control, and domain logic associated with a relatively fine-grained abstraction. Groups of these self-contained components are combined to build coarser-grained abstractions (with correspondingly more complex displays, perhaps panels and entire forms). These coarser-grained abstractions are gathered together to make programs.

In the PAC architecture, the lowest-level objects are likely to be subtypes of specific controls; for instance, a **Button** that encapsulates a bit of domain logic relating to a trigger or switch. Mid-level objects may descend from **UserControl** (essentially, an interface-less **Control**) and would encapsulate discrete chunks of business logic. Higher-level objects would likely descend from **Form** and are likely to encapsulate all the logic associated with a particular scenario or use-case.

In this example, we have a type descended from **Button** that knows whether it is on or off and a type descended from **Panel** that contains these **TwoState** buttons and knows if all the **TwoState**s within it are in state "On":

```
//:c13:PAC.cs
//Presentation-Abstraction-Control
using System.Drawing;
using System.Windows.Forms;
using System;

class TwoState : Button{
    static int instanceCounter = 0;
    int id = instanceCounter++;

    internal TwoState(){
        this.Text = State;
        System.EventHandler hndlr =
```

```
            new System.EventHandler(buttonClick);
        this.Click += hndlr;
    }

    bool state = true;
    public string State{
        get {
           return (state == true) ? "On" : "Off";
        }
        set{
           state = (value == "On") ? true : false;
           OnStateChanged();
        }
    }

    private void buttonClick(
      object sender, System.EventArgs e){
        changeState();
    }

    public void changeState(){
       state = !state;
       OnStateChanged();
    }

    public void OnStateChanged(){
        System.Console.WriteLine(
           "TwoState id " + id + " state changed");
        this.Text = State;
    }
}

class ChristmasTree : Panel{
   bool allOn;
   internal bool AllOn{
     get { return allOn; }
   }

   public ChristmasTree(){
      TwoState ts = new TwoState();
      ts.Location = new Point(10, 10);
```

```
      TwoState ts2 = new TwoState();
      ts2.Location = new Point(120, 10);
      Add(ts);
      Add(ts2);
      BackColor = Color.Green;
   }

   public void Add(TwoState c){
         Controls.Add(c);
         c.Click += new EventHandler(
            this.TwoClickChanged);
   }

   public void AddRange(TwoState[] ca){
      foreach(TwoState ts in ca){
         ts.Click += new EventHandler(
            this.TwoClickChanged);
      }
      Controls.AddRange(ca);
   }

   public void TwoClickChanged(
      Object src, EventArgs a){
        allOn = true;
        foreach(Control c in Controls){
           TwoState ts = c as TwoState;
           if(ts.State != "On"){
              allOn = false;
           }
        }
        if(allOn){
           BackColor = Color.Green;
        }else{
           BackColor = Color.Red;
        }
   }
}

class PACForm : Form {
    ChristmasTree p1 = new ChristmasTree();
    ChristmasTree p2 = new ChristmasTree();
```

```
   public PACForm(){
        ClientSize = new Size(450, 200);
        Text = "Events & Models";

        p1.Location = new Point(10,10);
        p2.Location = new Point(200, 10);
        Controls.Add(p1);
        Controls.Add(p2);
    }

    static void Main(){
        Application.Run(new PACForm());
    }
}

///:~
```

When run, if you set both the buttons within an individual **ChristmasTree** panel to "On," the **ChristmasTree**'s background color will become green, otherwise, the background color will be red. The **PACForm** knows nothing about the **TwoState**s within the **ChristmasTree**. We could (and indeed it would probably be logical) change **TwoState** from descending from **Button** to descending from **Checkbox** and **TwoState.State** from a **string** to a **bool** and it would make no difference to the **PACForm** that contains the two instances of **ChristmasTree**.

Presentation-Abstraction-Control is often the best architecture for working with .NET. It's killer advantage is that it provides an encapsulated component. A component is a software module that can be deployed and composed into other components *without source-code modification*. Visual Basic programmers have enjoyed the benefits of software components for more than a decade, with thousands of third-party components available. Visual Studio .NET ships with a few components that are at a higher level than just encapsulating standard controls, for instance, components which encapsulate ADO and another which encapsulates the Crystal Reports tools.

PAC components need not really be written as a single class; rather, a single **Control** class may provide a single **public** view of a more complex

namespace whose members are not visible to the outside world. This is called the *Façade* design pattern.

The problem with PAC is that it's hard work to create a decent component. Our **ChristmasTree** component is horrible – it doesn't automatically place the internal **TwoState**s reasonably, it doesn't resize to fit new **TwoState**s, it doesn't expose both logical and GUI events and properties… The list goes on and on. Reuse is the great unfulfilled promise of object orientation: "Drop a control on the form, set a couple of properties, and boom! You've got a payroll system." But the reality of development is that at least 90% of your time is absolutely controlled by the pressing issues of the current development cycle and there is little or no time to spend on details not related to the task at hand. Plus, it's difficult enough to create an effective GUI when you have direct access to your end-users; creating an effective GUI component that will be appropriate in situations that haven't yet arisen is almost impossible.

Nevertheless, Visual Studio .NET makes it so easy to create a reusable component (just compile your component to a .DLL and you can add it to the Toolbar!) that you should always at least consider PAC for your GUI architecture.

Note: control can't be in two control collections at once.

Model-View-Controller

Model-View-Controller, commonly referred to simply as "MVC," was the first widely known architectural pattern for decoupling the graphical user interface from underlying application logic. Unfortunately, many people confuse MVC with any architecture that separates presentation logic from domain logic. So when someone starts talking about MVC, it's wise to allow for quite a bit of imprecision.

In MVC, the Model encapsulates the system's logical behavior and state, the View requests and reflects that state on the display, and the Controller interprets low-level inputs such as mouse and keyboard strokes, resulting in commands to either the Model or the View.

MVC trades off a lot of static structure -- the definition of objects for each of the various responsibilities – for the advantage of being able to independently vary the view and the controller. This is not much of an advantage in Windows programs, where the view is always a bitmapped two-dimensional display and the controller is always a comination of a keyboard and a mouse-like pointing device. However, USB's widespread support has already led to interesting new controllers and the not-so-distant future will bring both voice and gesture control and "hybrid reality" displays to seamlessly integrate computer-generated data into real vision (e.g., glasses that superimpose arrows and labels on reality). If you happen to be lucky enough to be working with such advanced technologies, MVC may be just the thing. Even if not, it's worth discussing briefly as an example of decoupling GUI concerns taken to the logical extreme. In our example, our domain state is simply an array of Boolean values; we want the display to show these values as buttons and display, in the title bar, whether all the values are true or whether some are false:

```
//:c13:MVC.cs
using System;
using System.Windows.Forms;
using System.Drawing;

class Model{
   internal bool[] allBools;

   internal Model(){
      Random r = new Random();
      int iBools = 2 + r.Next(3);
      allBools = new bool[iBools];
      for(int i = 0; i < iBools; i++){
         allBools[i]  = r.NextDouble() > 0.5;
      }
   }
}

class View : Form{
   Model model;
   Button[] buttons;
```

```
   internal View(Model m, Controller c){
      this.model = m;

      int buttonCount = m.allBools.Length;
      buttons = new Button[buttonCount];
      ClientSize =
         new Size(300, 50 + buttonCount * 50);
      for(int i = 0; i < buttonCount; i++){
         buttons[i] = new Button();
         buttons[i].Location =
            new Point(10, 5 + i * 50);
         c.MatchControlToModel(buttons[i], i);
         buttons[i].Click +=
            new EventHandler(c.ClickHandler);
      }
      ReflectModel();
      Controls.AddRange(buttons);
   }

   internal void ReflectModel(){
      bool allAreTrue = true;
      for(int i = 0; i < model.allBools.Length; i++){
         buttons[i].Text =
            model.allBools[i].ToString();
         if(model.allBools[i] == false){
            allAreTrue = false;
         }
      }
      if(allAreTrue){
         Text = "All are true";
      }else{
         Text = "Some are not true";
      }
   }
}

class Controller{
   Model model;
   View view;

   Control[] viewComponents;
```

```
   Controller(Model m){
      model = m;
      viewComponents = new Control[m.allBools.Length];
   }

   internal void MatchControlToModel
    (Button b, int index){
     viewComponents[index] = b;
   }

   internal void AttachView(View v){
      this.view = v;
   }

   internal void ClickHandler
      (Object src, EventArgs ea){
      //Modify model in response to input
      int modelIndex =
        Array.IndexOf(viewComponents, src);
      model.allBools[modelIndex] =
        !model.allBools[modelIndex];
      //Have view reflect model
      view.ReflectModel();
   }

   public static void Main(){
      Model m = new Model();
      Controller c = new Controller(m);
      View v = new View(m, c);
      c.AttachView(v);
      Application.Run(v);
   }
}///:~
```

The **Model** class has an array of **bools** that are randomly set in the **Model** constructor. For demonstration purposes, we're exposing the array directly, but in real life the state of the **Model** would be reflected in its entire gamut of properties.

The **View** object is a **Form** that contains a reference to a **Model** and its role is simply to reflect the state of that **Model**. The **View()** constructor

lays out how this particular view is going to do that – it determines how many **bool**s are in the **Model**'s **allBools** array and initializes a **Button[]** array of the same size. For each **bool** in **allBools**, it creates a corresponding **Button,** and associates this particular aspect of the **Model**'s state (the index of this particular **bool**) with this particular aspect of the **View** (this particular **Button**). It does this by calling the **Controller**'s **MatchControlToModel( )** method and by adding to the **Button**'s **Click** event property a reference to the **Controller**'s **ClickHandler( )**. Once the **View** has initialized its **Controls,** it calls its own **ReflectModel( )** method.

View's **ReflectModel( )** method iterates over all the **bool**s in **Model**, setting the text of the corresponding button to the value of the Boolean**.** If all are **true**, the title of the **Form** is changed to "All are true," otherwise, it declares that "Some are false."

The **Controller** object ultimately needs references to both the **Model** and to the **View**, and the **View** needs a reference to both the **Model** and the **Controller**. This leads to a little bit of complexity in the initialization shown in the **Main( )** method; the **Model( )** is created with no references to anything (the **Model** is acted upon by the **Controller** and reflected by the **View**, the **Model** itself never needs to call methods or properties in those objects). The **Controller** is then created with a reference to the **Model**. The **Controller( )** constructor simply stores a reference to the **Model** and initializes an array of **Control**s to sufficient size to reflect the size of the **Model.allBools** array. At this point, the **Controller.View** reference is still null, since the **View** has not yet been initialized.

Back in the **Main( )** method, the just-created **Controller** is passed to the **View( )** constructor, which initializes the **View** as described previously. Once the **View** is initialized, the **Controller**'s **AttachView( )** method sets the reference to the **View**, completing the **Controller**'s initialization. (You could as easily do the opposite, creating a **View** with just a reference to the **Model**, creating a **Controller** with a reference to the **View** and the **Model**, and then finish the **View**'s initialization with an **AttachController( )** method.)

During the **View**'s constructor, it called the **Controller**'s **MatchControlToModel( )** method, which we can now see simply stores a reference to a **Button** in the **viewComponents[ ]** array.

The **Controller** is responsible for interpreting events and causing updates to the **Model** and **View** as appropriate. **ClickHandler( )** is called by the various **Button**s in the **View** when they are clicked. The originating **Control** is referenced in the **src** method argument, and because the index in the **viewComponents[ ]** array was defined to correspond to the index of the **Model**'s **allBools[ ]** array, we can learn what aspect of the **Model**'s state we wish to update by using the static method **Array.IndexOf( )**. We change the Boolean to its opposite using the ! operator and then, having changed the **Model**'s state, we call **View**'s **ReflectModel( )** method to keep everything in synchrony.

The clear delineation of duties in MVC is appealing – the **View** passively reflects the **Model**, the **Controller** mediates updates**,** and the **Model** is responsible only for itself. You can have many **View** classes that reflect the same **Model** (say, one showing a graph of values, the other showing a list) and dynamically switch between them. However, the structural complexity of MVC is a considerable burden and is difficult to "integrate" with the Visual Designer tool.

Layout

Now that we've discussed the various architectural options that should be of major import in any real GUI design discussion, we're going to move back towards the expedient "Do as we say, not as we do" mode of combining logic, event control, and visual display in the sample programs. It would simply consume too much space to separate domain logic into separate classes when, usually, our example programs are doing nothing but writing out simple lines of text to the console or demonstrating the basics of some simple widget.

Another area where the sample programs differ markedly from professional code is in the layout of **Control**s. So far, we have used the **Location** property of a **Control,** which determines the upper-left corner of **this Control** in the client area of its containing **Control** (or, in the

case of a **Form**, the Windows display coordinates of its upper-left corner).

More frequently, you will use the **Dock** and **Anchor** properties of a **Control** to locate a **Control** relative to one or more *edges* of the container in which it resides. These properties allow you to create **Control**s which properly resize themselves in response to changes in windows size.

In our examples so far, resizing the containing Form doesn't change the **Control** positions. That is because by default, **Control**s have an **Anchor** property set to the **AnchorStyles** values **Top** and **Left** (**AnchorStyles** are bitwise combinable). In this example, a button moves relative to the opposite corner:

```
//:c13:AnchorValues.cs
using System.Windows.Forms;
using System.Drawing;

class AnchorValues: Form{
   AnchorValues(){
      Button b = new Button();
      b.Location = new Point(10, 10);
      b.Anchor =
         AnchorStyles.Right | AnchorStyles.Bottom;
      Controls.Add(b);
   }

   public static void Main(){
      Application.Run(new AnchorValues());
   }
}///:~
```

If you combine opposite **AnchorStyles** (Left and Right, Top and Bottom) the **Control** will resize. If you specify **AnchorStyles**.**None**, the control will move half the distance of the containing area's change in size. This example shows these two types of behavior:

```
//:c13:AnchorResizing.cs
using System.Windows.Forms;
using System.Drawing;
```

```
class AnchorResizing: Form{
   AnchorResizing(){
      Button b = new Button();
      b.Location = new Point(10, 10);
      b.Anchor =
         AnchorStyles.Left
         | AnchorStyles.Right
         | AnchorStyles.Top;
      b.Text = "Left | Right | Top";
      Controls.Add(b);


      Button b2 = new Button();
      b2.Location = new Point(100, 10);
      b2.Anchor = AnchorStyles.None;
      b2.Text = "Not anchored";
      Controls.Add(b2);
   }

   public static void Main(){
      Application.Run(new AnchorResizing());
   }
}///:~
```

If you run this example and manipulate the screen, you'll see two undesirable behaviors: **b** can obscure **b2**, and **b2** can float off the page. Windows Forms' layout behavior trades off troublesome behavior like this for its straightforward model. An alternative mechanism based on cells, such as that used in HTML or some of Java's **LayoutManagers**, may be more robust in avoiding these types of trouble, but anyone who's tried to get a complex cell-based UI to resize the way they wish is likely to agree with Windows Forms' philosophy!

A property complementary to **Anchor** is **Dock**. The **Dock** property moves the control flush against the specified edge of its container, and resizes the control to be the same size as that edge. If more than one control in a client area is set to **Dock** to the same edge, the controls will layout side-by-side in the *reverse of the order* in which they were added to the containing **Controls** array (their *reverse z-order*). The **Dock** property overrides the **Location** value. In this example, two buttons are created and docked to the left side of their containing **Form**.

```
//:c13:Dock.cs
using System.Windows.Forms;
using System.Drawing;

class Dock: Form{
   Dock(){
      Button b1 = new Button();
      b1.Dock = DockStyle.Left;
      b1.Text = "Button 1";
      Controls.Add(b1);


      Button b2 = new Button();
      b2.Dock = DockStyle.Left;
      b2.Text = "Button2";
      Controls.Add(b2);
   }

   public static void Main(){
      Application.Run(new Dock());
   }
}///:~
```

When you run this, you'll see that **b** appears *to the right* of **b2** because it was added to **Dock**'s **Controls** *before* **b2.**

DockStyle.Fill specifies that the **Control** will expand to fill the client area from the center to the limits allowed by other **Dock**ed **Controls**. **DockStyle.Fill** will cover non-**Dock**ed **Controls** that have a lower z-order, as this example shows:

```
//:c13:DockFill.cs
using System.Windows.Forms;
using System.Drawing;

class DockFill: Form{
   DockFill(){
      //Lower z-order
      Button visible = new Button();
      visible.Text = "Visible";
      visible.Location = new Point(10, 10);
      Controls.Add(visible);
```

```
      //Will cover "Invisible"
      Button docked = new Button();
      docked.Text = "Docked";
      docked.Dock = DockStyle.Fill;
      Controls.Add(docked);


      //Higher z-order, gonna' be invisible
      Button invisible = new Button();
      invisible.Text = "Invisible";
      invisible.Location = new Point(100, 100);
      Controls.Add(invisible);
   }

   public static void Main(){
      Application.Run(new DockFill());
   }
}///:~
```

Developing complex layouts that lay themselves out properly when resized is a challenge for any system. Windows Forms' straightforward model of containment, **Location, Anchor**, and **Dock** is especially suited for the PAC GUI architecture described previously. Rather than trying to create a monolithic chunk of logic that attempts to resize and relocate the hundreds or dozens of widgets that might comprise a complex UI, the PAC architecture would encapsulate the logic within individual custom controls.

Non-Code Resources

Windows Forms wouldn't be much of a graphical user interface library if it did not support graphics and other media. But while it's easy to specify a **Button**'s look and feel with only a few lines of code, images are inherently dependent on binary data storage.

It's not surprising that you can load an image into a Windows Form by using a **Stream** or a filename, as this example demonstrates:

```
//:c13:SimplePicture.cs
//Loading images from file system
```

```
using System;
using System.IO;
using System.Windows.Forms;
using System.Drawing;

class SimplePicture : Form {
   public static void Main(string[] args){
      SimplePicture sp = new SimplePicture(args[0]);
      Application.Run(sp);
   }

   SimplePicture(string fName){
        PictureBox pb = new PictureBox();
        pb.Image = Image.FromFile(fName);
        pb.Dock = DockStyle.Fill;
        pb.SizeMode = PictureBoxSizeMode.StretchImage;
        Controls.Add(pb);

        int imgWidth = pb.Image.Width;
        int imgHeight = pb.Image.Height;

        this.ClientSize =
           new Size(imgWidth, imgHeight);
   }

}///:~
```

The **Main( )** method takes the first command-line argument as a path to an image (for instance: "SimplePicture c:\windows\clouds.bmp") and passes that path to the **SimplePicture( )** constructor. The most common **Control** used to display a bitmap is the **PictureBox** control, which has an **Image** property. The static method **Image.FromFile( )** generates an **Image** from the given path (there is also an **Image.FromStream( )** static method which provides general access to all the possible sources of image data).

The **PictureBox**'s **Dock** property is set to **Dockstyle.Fill** and the **ClientSize** of the form is set to the size of the **Image**. When you run this program, the **SimplePicture** form will start at the same size of the image. Because **pb.SizeMode** was set to **StretchImage**, however, you

can resize the form and the image will stretch or shrink appropriately. Alternate **PictureBoxSizeMode** values are **Normal** (which clips the **Image** to the **PictureBox**'s size), **AutoSize** (which resizes the **PictureBox** to accommodate the **Image**'s size), and **CenterImage.**

Loading resources from external files is certainly appropriate in many circumstances, especially with isolated storage (#ref#), which gives you a per-user, consistent virtual file system. However, real applications which are intended for international consumption require many resources localized to the current culture – labels, menu names, and icons may all have to change. The .NET Framework provides a standard model for efficiently storing such resources and loading them. Momentarily putting aside the question of how such resources are created, retrieving them is the work of the **ResourceManager** class. This example switches localized labels indicating "man" and "woman."

```
//:c13:International.cs
using System;
using System.Drawing;
using System.Resources;
using System.Globalization;
using System.Windows.Forms;

class International : Form{
   ResourceManager rm;
   Label man;
   Label woman;

   public International(){
      rm = new ResourceManager(typeof(International));

      RadioButton eng = new RadioButton();
      eng.Checked = true;
      eng.Location = new Point(10, 10);
      eng.Text = "American";
      eng.CheckedChanged +=
         new EventHandler(LoadUSResources);

      RadioButton swa = new RadioButton();
      swa.Location = new Point(10, 30);
```

```
      swa.Text = "Swahili";
      swa.CheckedChanged +=
         new EventHandler(LoadSwahiliResources);

      man = new Label();
      man.Location = new Point(10, 60);
      man.Text = "Man";

      woman = new Label();
      woman.Location = new Point(10, 90);
      woman.Text = "Woman";

      Controls.AddRange(new Control[] {
         eng, swa, man, woman});

      Text = "International";
   }

   public void LoadUSResources
   (Object src, EventArgs a){
     if ( ((RadioButton)src).Checked == true) {
         ResourceSet rs =
         rm.GetResourceSet(
            new CultureInfo("en-US"), true, true);
         SetLabels(rs);
      }
   }

   public void LoadSwahiliResources
   (Object src, EventArgs a){
      if (((RadioButton)src).Checked == true) {
         ResourceSet rs =
         rm.GetResourceSet(
            new CultureInfo("sw"), true, true);
         SetLabels(rs);
      }
   }

   private void SetLabels(ResourceSet rs){
      man.Text = rs.GetString("Man");
      woman.Text = rs.GetString("Woman");
```

```
    }

    public static void Main(){
       Application.Run(new International());
    }
}
```

Here, we wish to create an application which uses labels in a local culture (a culture is more specific than a language; for instance, there is a distinction between the culture of the United States and the culture of the United Kingdom). The basic references we'll need are to **ResourceManager rm**, which we'll load to be culture-specific, and two labels for the words "Man" and "Woman."

The first line of the **International** constructor initializes **rm** to be a resource manager for the specified type. A **ResourceManager** is associated with a specific type because .NET uses a "hub and spoke" model for managing resources. The "hub" is the assembly that contains the code of a specific type. The "spokes" are zero or more *satellite assemblies* that contain the resources for specific cultures, and the **ResourceManager** is the link that associates a type (a "hub") with its "spokes".

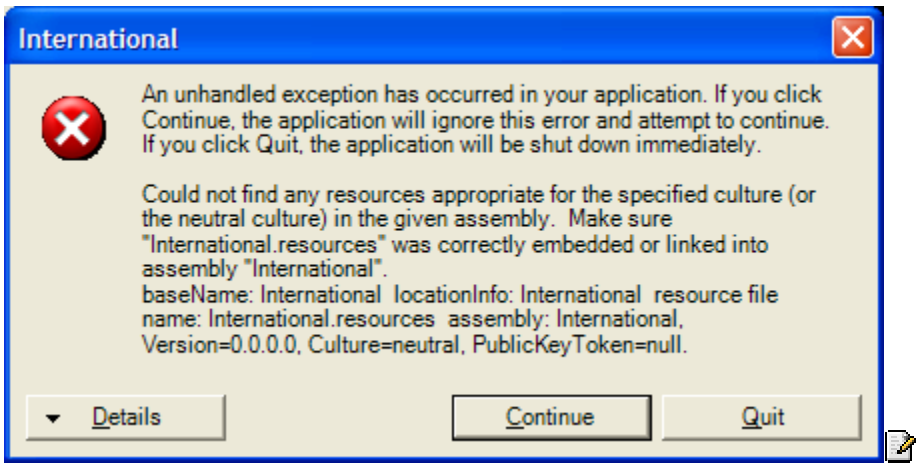
International( ) then shows the use of radio buttons in Windows Forms. The model is simple: all radio buttons within a container are mutually exclusive. To make multiple sets of radio buttons within a single form, you can use a **GroupBox** or **Panel**. **International** has two **RadioButton**s, **eng** has its **Checked** property set to **true**, and, when that property changes, the **LoadEnglishResources** method will be called. **RadioButton swa** is similarly configured to call **LoadSwahiliResources( )** and is not initially checked. By default, the **man** and **woman** labels are set to a hard-coded value.

The **Load*xxx*Resources( )** methods are similar; they check if their source **RadioButton** is checked (since they are handling the **CheckedChange** event, the methods will be called when their source becomes de-checked as well). If their source *is* set, the **ResourceManager** loads one of the "spoke" **ResourceSet** objects. The **ResourceSet** is associated with a particular **CultureInfo** instance, which is initialized with a *language tag* **string** compliant with IETF RFC

1766 (you can find a list of standard codes in the .NET Framework documentation and read the RFC at http://www.ietf.org/rfc/rfc1766.txt). The **GetResourceSet( )** method also takes two **bools**, the first specifying if the **ResourceSet** should be loaded if it does not yet exist in memory, and the second specifying if the **ResourceManager** should try to load "parents" of the culture if the specified **CultureInfo** does not work; both of these **bool**s will almost always be **true**.

Once the **ResourceSet** is retrieved, it is used as a parameter to the **SetLabels( )** method. **SetLabels( )** uses **ResourceSet.GetString( )** to retrieve the appropriate culture-specific **string** for the specified key and sets the associated **Label.Text**. **ResourceSet**'s other major method is **GetObject( )** which can be used to retrieve any type of resource.

We've not yet created the satellite assemblies which will serve as the "spokes" to our **International** "hub," but it is interesting to run the program in this state. If you run the above code and click the "Swahili" radio button, you will see this dialog:


International
An unhandled exception has occurred in your application. If you click Continue, the application will ignore this error and attempt to continue. If you click Quit, the application will be shut down immediately.
Could not find any resources appropriate for the specified culture (or the neutral culture) in the given assembly. Make sure "International.resources" was correctly embedded or linked into assembly "International". baseName: International locationInfo: International resource file name: International.resources assembly: International, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null.
Details
Continue
Quit

This is not a dialog you'd ever want an end-user to see and a real application's exception handling would hide it, but it's an interesting example of the kind of behavior that you could potentially include in your own components to aid 3rd party developers during debugging.

Creating Satellite Assemblies

For your satellite assembly to work, you must follow naming (including capitalization) and directory conventions. First, you will create a new subdirectory for each culture you wish to support and named with the culture's language tag. If you compiled **International.exe** in the directory c:\tic\chap13, you will create c:\tic\chap13\sw and c:\tic\chap13\en-US.

In the \sw subdirectory, create a file with these contents:

```
Man=mwanamume
Woman=mwanamke
```

And save the file with a .txt extension (say, as "Swahili.txt"). Use the command-line resource generator tool to turn this file into a binary resource file that follows the naming convention ***MainAssembly*.*languagetag*.resources**. For this example, the command is:

```
resgen swahili.txt International.sw.resources
```

The **.resources** file now has to be converted into a satellite assembly named ***MainAssembly*.resources.dll**. This is done with the *assembly linker* tool al. Both of these lines should be typed as a single command:

```
al /t:lib /embed:International.sw.resources
/culture:sw /out:International.resources.dll
```

The resulting .DLL should still be in the \sw subdirectory. Do the same process in the \en-US directory after creating an appropriate text file:

```
resgen american.txt International.en-US.resources

al /t:lib /embed:International.en-US.resources
/culture:en-US /out:International.resources.dll
```

Switch back to the parent directory and run **International**. Now, when you run the program, the **man** and **woman** labels should switch between Swahili and American in response to the radio buttons.

Constant Resources

While culturally appropriate resources use satellite assemblies, it may be the case that you wish to have certain resources such as graphics and icons embedded directly in the main assembly. Using graphics as resources is a little more difficult than using text because you must use a utility class to generate the resource file. Here's an example command-line class that takes two command-line arguments: the name of a graphics file and the name of the desired resources file:

```
//:c13:GrafResGen.cs
//Generates .resource file from a graphics file
//Usage: GrafResGen [inputFile] [outputFile]
using System.IO;
using System.Resources;
using System.Drawing;

class GrafResGen {
   GrafResGen(
     string name, Stream inStr, Stream outStr){
      ResourceWriter rw = new ResourceWriter(outStr);

      Image img = new Bitmap(inStr);
      rw.AddResource(name, img);
      rw.Generate();
   }

   public static void Main(string[] args){
      FileStream inF = null;
      FileStream outF = null;
      try {
         string name = args[0];
         inF = new FileStream(name, FileMode.Open);
         string outName = args[1];
         outF =
            new FileStream(outName, FileMode.Create);
         GrafResGen g =
            new GrafResGen(name, inF, outF);
      } finally {
         inF.Close();
```

```
            outF.Close();
        }
    }
}//:~
```

A **ResourceWriter** generates binary **.resource** files to a given **Stream**. A **ResXResourceWriter** (not demonstrated) can be used to create an XML representation of the resources that can then be compiled into a binary file using the **resgen** process described above (an XML representation is not very helpful for binary data, so we chose to use a **ResourceWriter** directly).

To use this program, copy an image to the local directory and run:

```
GrafResGen someimage.jpg ConstantResources.resources
```

This will generate a binary resources file that we'll embed in this example program:

```
//:c13:ConstantResources.cs
/*
Compile with: csc /res:ConstantResources.resources
  ConstantResources.cs
*/
//Loads resources from the current assembly
using System.Resources;
using System.Drawing;
using System.Windows.Forms;


class ConstantResources:Form{
     ConstantResources(){
        PictureBox pb = new PictureBox();
        pb.Dock = DockStyle.Fill;
        Controls.Add(pb);

        ResourceManager rm =
           new ResourceManager(this.GetType());
        pb.Image =
          (Image) rm.GetObject("someimage.jpg");
     }

     public static void Main(){
```

```
        Application.Run(new ConstantResources());
    }
}//:~
```

The code in **ConstantResources** is very similar to the code used to load cultural resources from satellites, but without the **GetResourceSet( )** call to load a particular satellite. Instead, the **ResourceManager** looks for resources associated with the **ConstantResources** type. Naturally, those are stored in the **ConstantResources.resources** file generated by the **GrafResGen** utility just described. For the **ResourceManager** to find this file, though, the resource file must be linked into the main assembly in this manner:

```
csc /res:ConstantResources.res ConstantResources.cs
```

Assuming that the resources have been properly embedded into the **ConstantResources.exe** assembly, the **ResourceManager** can load the "**someimage.jpg**" resource and display it in the **PictureBox pb**.

What About the XP Look?

If you have been running the sample programs under Windows XP, you may have been disappointed to see that **Control**s do not automatically support XP's graphical themes. In order to activate XP-themed controls, you must set your **Control**'s **FlatStyle** property to **FlatStyle.System** and specify that your program requires Microsoft's **comctl6** assembly to run. You do that by creating another type of non-code resource for your file: a manifest. A manifest is an XML document that specifies all sorts of meta-information about your program: it's name, version, and so forth. One thing you can specify in a manifest is a dependency on another assembly, such as **comctl6**. To link to **comctl6**, you'll need a manifest of this form:

```
<!-- XPThemed.exe.manifest -->
<?xml
   version="1.0" encoding="UTF-8" standalone="yes"
?>
<assembly
  xmlns="urn:schemas-microsoft-com:asm.v1"
  manifestVersion="1.0">
   <assemblyIdentity
```

```
      type="win32"
      name="Thinkingin.Csharp.C13.XPThemes"
      version="1.0.0.0"
      processorArchitecture = "X86"
   />
   <description>Demonstrate XP Themes</description>
   <!-- Link to comctl6 -->
   <dependency>
      <dependentAssembly>
         <assemblyIdentity
           type="win32"
           name="Microsoft.Windows.Common-Controls"
           version="6.0.0.0"
           processorArchitecture="X86"
           publicKeyToken="6595b64144ccf1df"
           language="*"
         />
     </dependentAssembly>
  </dependency>
</assembly>
```

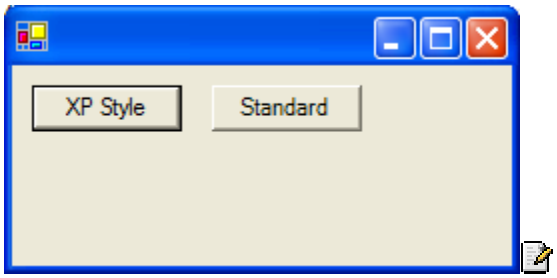
The manifest file is an XML-formatted source of meta-information about your program. In this case, after specifying our own **assemblyIdentity,** we specify the dependency on **Common-Controls** version 6.

Name this file *programName***.exe.manifest** and place it in the same directory as your program. If you do, the .NET Runtime will automatically give the appropriate **Control**s in your program XP themes. Here's an example program:

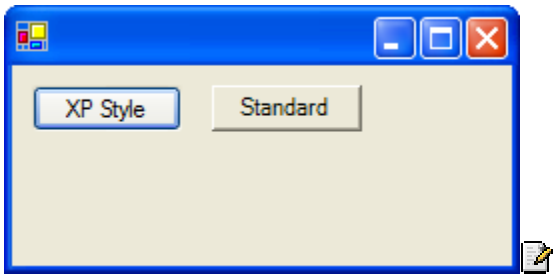
```
//:c13:XPThemed.cs
using System.Windows.Forms;
using System.Drawing;

class XPThemed: Form{
   XPThemed(){
      ClientSize = new Size(250, 100);
      Button b = new Button();
      b.Text = "XP Style";
      b.Location = new Point(10, 10);
      b.FlatStyle = FlatStyle.System;
      Controls.Add(b);
```

```
        Button b2 = new Button();
        b2.Text = "Standard";
        b2.Location = new Point(100, 10);
        Controls.Add(b2);
    }

    public static void Main(){
        Application.Run(new XPThemed());
    }
}///:~
```

When run without an appropriate manifest file, both buttons will have a default gray style:



When **XPThemed.exe.manifest** is available, **b** will use the current XP theme, while **b2,** whose **FlatStyle** is the default **FlatStyle.Standard**, will not.



Fancy Buttons

In addition to creating theme-aware buttons, it is an easy matter to create buttons that have a variety of graphical features and that change their appearance in response to events. In order to run this example program,

you'll have to have four images in the active directory (in the example code, they're assumed to be named "tic.gif", "away.gif","in.gif", and "hover.gif").

```
//:c13:ButtonForm.cs
///Demonstrates various types of buttons
using System.Windows.Forms;
using System;
using System.Collections;
using System.Drawing;

class ButtonForm : Form {

   ButtonForm() {
      ClientSize = new System.Drawing.Size(400, 200);
      Text = "Buttons, in all their glory";

      Button simple = new Button();
      simple.Text = "Simple";
      simple.Location = new Point(10, 10);

      Button image = new Button();
      image.Image = Image.FromFile(".\\TiC.gif");
      image.Text = "Text";
      image.Location = new Point(120, 10);

      Button popup = new Button();
      popup.Location = new Point(230, 10);
      popup.Text = "Popup";
      popup.FlatStyle = FlatStyle.Popup;

      FlyOverButton flyOver =
         new FlyOverButton("Away", "In", "Hovering");
      flyOver.Location = new Point(10, 40);

      FlyOverImages flyOverImages =
         new FlyOverImages
         (".\\away.gif",".\\in.gif",".\\hover.gif");
      flyOverImages.Location = new Point(230, 40);

      Controls.AddRange(new Control[]{
```

```
            simple, image, popup, flyOver, flyOverImages}
         );
      }

      public static void Main() {
         Application.Run(new ButtonForm());
      }
   }

   class FlyOverButton : Button {
      string away;
      string inStr;
      string hover;
      internal FlyOverButton(
         string away, string inStr, string hover) {
         this.away = away;
         this.inStr = inStr;
         this.hover = hover;
         FlatStyle = FlatStyle.Popup;
         Text = away;
         MouseEnter += new EventHandler(OnMouseEnter);
         MouseHover += new EventHandler(OnMouseHover);
         MouseLeave += new EventHandler(OnMouseLeave);
      }
      private void OnMouseEnter(
         object sender, System.EventArgs args) {
         ((Control)sender).Text = inStr;
      }

      private void OnMouseHover(
         object sender, System.EventArgs args) {
         ((Control)sender).Text = hover;
      }

      private void OnMouseLeave(
         object sender, System.EventArgs args) {
         ((Control)sender).Text = away;
      }
   }

   class FlyOverImages : Button {
```

```
    internal FlyOverImages(
       string away, string inStr, string hover) {
       ImageList = new ImageList();
       ImageList.Images.Add(Image.FromFile(away));
       ImageList.Images.Add(Image.FromFile(inStr));
       ImageList.Images.Add(Image.FromFile(hover));
       FlatStyle = FlatStyle.Popup;
       ImageIndex = 0;
       MouseEnter += new EventHandler(OnMouseEnter);
       MouseHover += new EventHandler(OnMouseHover);
       MouseLeave += new EventHandler(OnMouseLeave);
    }
    private void OnMouseEnter(
       object sender, System.EventArgs args) {
       ((Button)sender).ImageIndex = 1;
    }

    private void OnMouseHover(
       object sender, System.EventArgs args) {
       ((Button)sender).ImageIndex = 2;
    }

    private void OnMouseLeave(
       object sender, System.EventArgs args) {
       ((Button)sender).ImageIndex = 0;
    }
}///:~
```

The first button created and placed on the form is **simple** and it's appearance and behavior should be familiar. The second button **image**, sets its **Image** property from an **Image** loaded from a file. The same **Image** is displayed at all times; if the **Text** property is set, the label will be drawn *over* the **Image**.

The third button **popup** has a **FlatStyle** of **FlatStyle.Popup**. This button appears flat until the mouse passes over it, at which point it is redrawn with a 3-D look.

The fourth and fifth buttons require more code and so are written as their own classes: **FlyOverButton** and **FlyOverImages**. The **FlyOverButton** is a regular button, but has event handlers for

MouseEnter, **MouseHover**, and **MouseLeave** which set the **Text** property as appropriate.

FlyOverImages takes advantage of the **ImageList** property of **Button**. Like **FlyOverButton**, **FlyOverImages** uses mouse events to change the image displayed on the button, but instead of manipulating the **Image** property, it sets the **ImageIndex** property, which correspond indices in the **ImageList** configured in the **FlyOverImages( )** constructor.

Tooltips

The one "fancy" thing that the previous example did not show is probably the one you most expect – the help text that appears when the mouse hovers over a control for more than a few moments. Such tooltips are, surprisingly, not a property of the **Control** above which they appear, but rather are controlled by a separate **ToolTip** object. This would seem to violate a design rule-of-thumb: objects should generally contain a navigable reference to all objects externally considered associated. As a user or programmer, one would definitely consider the tooltip to be "part of" what distinguishes one control from another, so one should expect a **Tooltip** property in **Control**. Another surprise is that the **ToolTip** does not conform to the containment model of Windows Forms, it is not placed within the **Controls** collection of another **Control**. This is an example of how even the best-designed libraries (and Windows Forms is top-notch) contain inconsistencies and quirks; while it can be very helpful to study the design of a good library to aid your design education, all libraries contain questionable choices.

Adding a **ToolTip** to a **Control** requires that a reference to the **Control** and its desired text be passed to an instance of **ToolTip** (which presumably maintains an internal **IDictionary,** which begs the question of why a **ToolTip** instance is required rather than using a static method). Here's an example that shows the basic use of a **ToolTip**:

```
//:c13:TooltipDisplay.cs
using System;
using System.Drawing;
using System.Windows.Forms;
```

```
class TooltipDisplay : Form{
   TooltipDisplay(){
      Button b = new Button();
      b.Text = "Button";
      b.Location = new Point(10, 10);
      Controls.Add(b);

      ToolTip t = new ToolTip();
      t.SetToolTip(b, "Does nothing");
   }

   public static void Main(){
      Application.Run(new TooltipDisplay());
   }
}
```

Displaying & Editing Text

One of the most common tasks for a GUI is displaying formatted text. In Windows Forms, formatted text is the realm of the **RichTextBox**, which displays and manipulates text in Rich Text Format. The details of the RTF syntax are thankfully hidden from the programmer, text appearance is manipulated using various **Selection*xxx*** properties, which manipulate the chosen substring of the total **RichTextBox.Text**. You can even, if you're so inclined, get and set the full RTF text (which is actually helpful when dragging-and-dropping from, say, Word. Drag-and-drop is covered later in this chapter.)

This example allows you to add arbitrary text to a **RichTextBox** with various formatting options chosen semi-randomly:

```
//:c13:TextEditing.cs
///Demonstrates the TextBox and RichTextBox controls
using System.Windows.Forms;
using System;
using System.Collections;
using System.Drawing;

class TextEditing : Form {
    TextBox tb;
```

```
RichTextBox rtb;
Random rand = new Random();

TextEditing() {

    ClientSize = new Size(450, 400);
    Text = "Text Editing";

    tb = new TextBox();
    tb.Text = "Some Text";
    tb.Location = new Point(10, 10);

    Button bold = new Button();
    bold.Text = "Bold";
    bold.Location = new Point(350, 10);
    bold.Click += new EventHandler(bold_Click);

    Button color = new Button();
    color.Text = "Color";
    color.Location = new Point(350, 60);
    color.Click += new EventHandler(color_Click);

    Button size = new Button();
    size.Text = "Size";
    size.Location = new Point(350, 110);
    size.Click += new EventHandler(size_Click);

    Button font = new Button();
    font.Text = "Font";
    font.Location =new Point(350, 160);
    font.Click += new EventHandler(font_Click);

    rtb = new RichTextBox();
    rtb.Location = new Point(10, 50);
    rtb.Size = new Size(300, 180);

    Controls.AddRange(
       new System.Windows.Forms.Control[]{
          tb, rtb, bold, color, size, font});
}
```

```
private void AddAndSelectText() {
    string newText = tb.Text + "\n";
    int insertionPoint = rtb.SelectionStart;
    rtb.AppendText(newText);
    rtb.SelectionStart = insertionPoint;
    rtb.SelectionLength = newText.Length;
}

private void ResetSelectionAndFont() {
    /* Setting beyond end of textbox places
     insertion at end of text */
    rtb.SelectionStart = Int16.MaxValue;
    rtb.SelectionLength = 0;
    rtb.SelectionFont =
       new Font("Verdana", 10, FontStyle.Regular);
    rtb.SelectionColor = Color.Black;
}
private void bold_Click(
   object sender, System.EventArgs e) {
    AddAndSelectText();
    rtb.SelectionFont =
       new Font("Verdana", 10, FontStyle.Bold);
    ResetSelectionAndFont();
}

private void color_Click(
   object sender, System.EventArgs e) {
    AddAndSelectText();
    rtb.SelectionColor =
       (rand.NextDouble()) > 0.5 ?
          Color.Red : Color.Blue;
    ResetSelectionAndFont();
}

private void size_Click(
   object sender, System.EventArgs e) {
    AddAndSelectText();
    int fontSize = 8 + rand.Next(10);
    rtb.SelectionFont =
       new Font("Verdana", fontSize,
```

```
                    FontStyle.Regular);
        ResetSelectionAndFont();
    }

    private void font_Click(
       object sender, System.EventArgs e) {
        AddAndSelectText();
        FontFamily[] families = FontFamily.Families;
        int iFamily = rand.Next(families.Length);
        rtb.SelectionFont =
           new Font(families[iFamily], 10,
                    FontStyle.Regular);
        ResetSelectionAndFont();
    }

    static void Main() {
        Application.Run(new TextEditing());
    }
}///:~
```

Everything in the **TextEditing( )** constructor should be familiar: a number of **Button**s are created, event handlers attached, and a **TextBox** and **RichTextBox** are placed on the **Form** as well.

The methods **AddAndSelectText( )** and **ResetSelectionAndFont( )** are used by the various event handlers. In **AddAndSelectText( )** the text to be inserted is taken from the **TextBox tb** and a newline added. The current **rtb.SelectionStart** is remembered, the new text appended to the **RichTextBox,** and the selection is set to begin with the remembered **insertionPoint** and **SelectionLength** to the length of the inserted text.

ResetSelectionAndFont( ) sets the insertion point at the end of the text by giving it an impossibly high value. The selection is reset to use the default font (10 pt. Verdana in black) using the appropriate properties.

The various event handlers call **AddAndSelectText( )** and then manipulate various aspects of the selected text – different sizes, colors, and fonts are randomly chosen.

Linking Text

In the past decade, hypertext has gone from an esoteric topic to probably the dominant form of human-computer interaction. However, incorporating text links into a UI has been a big challenge. Windows Forms changes that with its **LinkLabel** control. The **LinkLabel** has powerful support for linking, allowing any number of links within the label area. The **LinkLabel** facilitates the creation of even complex linking semantics, such as the XLink standard (www.w3.org/TR/xlink/).

While it's possible to use a **LinkLabel** to activate other Windows Forms behavior, the most common use is likely to be activating the full-featured Web browser. To do that, we need to introduce the **Process** class from the **System.Diagnostics** namespace. The **Process** class provides thorough access to local and remote processes, but the core functionality is starting a local process, i.e., launching another application while your application continues to run (or shuts down – the launched process is independent of your application). There are three overloaded versions of **Process.Start( )** that provide various degrees of control over the launched application. The most basic **Process.Start( )** method just takes a **string** and uses the OS's underlying mechanism to determine the appropriate way to run the request; if the string specifies a non-executable file, the extension may be associated with a program and, if so, that program will open it. For instance, **ProcessStart("Foo.cs")** will open the editor associated with the **.cs** extension.

The most advanced **Process.Start( )** takes a **ProcessStartInfo** info. **ProcessStartInfo** contains properties for setting environment variables, whether a window should be shown and in what style, input and output redirection, etc. This example uses the third overload of **Process.Start( )**, which takes an application name and a string representing the command-line arguments, to launch Internet Explorer and surf to www.ThinkingIn.Net.

```
//:c13:LinkLabelDemo.cs
//Demonstrates the LinkLabel
using System;
using System.Drawing;
using System.Diagnostics;
```

```
using System.Windows.Forms;

class LinkLabelDemo : Form {
   LinkLabelDemo(){
      LinkLabel label1 = new LinkLabel();
      label1.Text = "Download Thinking in C#";
      label1.Links.Add(9, 14,
         "http://www.thinkingin.net/");
      label1.LinkClicked +=
         new LinkLabelLinkClickedEventHandler(
            InternetExplorerLaunch);
      label1.Location = new Point(10, 10);
      label1.Size = new Size(160, 30);
      Controls.Add(label1);

      LinkLabel label2 = new LinkLabel();
      label2.Text = "Show message";
      label2.Links.Add(0, 4, "Foo");
      label2.Links.Add(5, 8, "Bar");
      label2.LinkClicked +=
         new LinkLabelLinkClickedEventHandler(
            MessageBoxShow);
      label2.Location = new Point(10, 60);
      Controls.Add(label2);
   }

   public void InternetExplorerLaunch
      (object src, LinkLabelLinkClickedEventArgs e){
      string url = (string) e.Link.LinkData;
      Process.Start("IExplore.exe", url);
      e.Link.Visited = true;
   }

   public void MessageBoxShow
      (object src, LinkLabelLinkClickedEventArgs e){
      string msg = (string) e.Link.LinkData;
      MessageBox.Show(msg);
      e.Link.Visited = true;
   }

   public static void Main(){
```

```
        Application.Run(new LinkLabelDemo());
    }
}//:~
```

Not all the **LinkLabel.Text** need be a link; individual links are added to the **Links** collection by specifying an offset and the length of the link. If you do not need any information other than the fact that the link was clicked, you do not need to include the third argument to **Links.Add( )**, but typically you will store some data to be used by the event handler. In the example, the phrase "Thinking in C#" is presented underlined, in the color of the **LinkColor** property (by default, this is set by the system and is usually blue). This link has as its associated data, a URL. **Label2** has two links within it, one associated with the **string** "foo" and the other with "bar."

While a **LinkLabel** can have many links, all the links share the same event handler (of course, it's a multicast delegate, so you can add as many methods to the event handling chain as desired, but you cannot directly associated a specific event-handling method with a specific link). The **Link** is passed to the event handler via the event arguments, so the delegate is the descriptively named **LinkLabelLinkClickedEventHandler**. The **LinkData** (if it was specified in the **Link**'s constructor) can be any **object**. In the example, we downcast the **LinkData** to **string**.

The **InternetExplorerLaunch( )** method uses **Process.Start( )** to launch Microsoft's Web browser. **MessageBoxShow( )** demonstrates the convenient **MessageBox** class, which pops up a simple alert dialog. At the end of the event handlers, the appropriate **Link** is set to **Visited**, which redraws the link in the **LinkLabel**.**VisitedLinkColor**.

Checkboxes and RadioButtons

As briefly mentioned in the **International** example, radio buttons in Windows Forms have the simple model of being mutually exclusive within their containing **Controls** collection. Sometimes it is sufficient to just plunk some radio buttons down on a **Form** and be done with it, but

usually you will use a **Panel** or **GroupBox** to contain a set of logically related radio buttons (or other controls). Usually, a set of related **RadioButton**s should have the same event handler since generally the program needs to know "Which of the radio buttons in the group is selected?" Since **EventHandler** delegates pass the source **object** as the first parameter in their arguments, it is easy for a group of buttons to share a single delegate method and use the **src** argument to determine which button has been activated. The use of a **GroupBox** and this form of sharing a delegate method is demonstrated in the next example.

CheckBox controls are not mutually exclusive; any number can be in any state within a container. **CheckBox**es cycle between two states (**CheckState.Checked** and **CheckState.Unchecked**) by default, but by setting the **ThreeState** property to true, can cycle between **Checked**, **Unchecked**, and **CheckState.Indeterminate**.

This example demonstrates a standard **CheckBox**, one that uses an **Image** instead of text (like **Button**s and many other **Control**s, the default appearance can be changed using a variety of properties), a three-state **CheckBox**, and grouped, delegate-sharing **RadioButton**s:

```
//:c13:CheckAndRadio.cs
//Demonstrates various types of buttons
using System.Windows.Forms;
using System;
using System.Collections;
using System.Drawing;

class CheckAndRadio : Form {
   CheckAndRadio() {
      ClientSize = new System.Drawing.Size(400, 200);
      Text = "Checkboxes and Radio Buttons";

      CheckBox simple = new CheckBox();
      simple.Text = "Simple";
      simple.Location = new Point(10, 10);
      simple.Click +=
         new EventHandler(OnSimpleCheckBoxClick);

      CheckBox image = new CheckBox();
```

```
        image.Image = Image.FromFile(".\\TiC.gif");
        image.Location = new Point(120, 10);
        image.Click +=
           new EventHandler(OnSimpleCheckBoxClick);

        CheckBox threeState = new CheckBox();
        threeState.Text = "Three state";
        threeState.ThreeState = true;
        threeState.Location = new Point(230, 10);
        threeState.Click +=
           new EventHandler(OnThreeStateCheckBoxClick);

        Panel rbPanel = new Panel();
        rbPanel.Location = new Point(10, 50);
        rbPanel.Size = new Size(420, 50);
        rbPanel.AutoScroll = true;

        RadioButton f1 = new RadioButton();
        f1.Text = "Vanilla";
        f1.Location = new Point(0, 10);
        f1.CheckedChanged +=
           new EventHandler(OnRadioButtonChange);
        RadioButton f2 = new RadioButton();
        f2.Text = "Chocolate";
        f2.Location = new Point(140, 10);
        f2.CheckedChanged +=
           new EventHandler(OnRadioButtonChange);
        RadioButton f3 = new RadioButton();
        f3.Text = "Chunky Monkey";
        f3.Location = new Point (280, 10);
        f3.CheckedChanged +=
           new EventHandler(OnRadioButtonChange);
        f3.Checked = true;

        rbPanel.Controls.AddRange(
           new Control[]{ f1, f2, f3});

        Controls.AddRange(
           new Control[]{
             simple, image, threeState, rbPanel});
    }
```

```
   private void OnSimpleCheckBoxClick(
      object sender, EventArgs args){
      CheckBox cb = (CheckBox) sender;
      System.Console.WriteLine(
         cb.Text + " is " + cb.Checked);
   }

   private void OnThreeStateCheckBoxClick(
      object sender, EventArgs args){
      CheckBox cb = (CheckBox) sender;
      System.Console.WriteLine(
         cb.Text + " is " + cb.CheckState);
   }

   private void OnRadioButtonChange(
      object sender, EventArgs args){
      RadioButton rb = (RadioButton) sender;
      if (rb.Checked == true)
         System.Console.WriteLine(
            "Flavor is " + rb.Text);
   }

   public static void Main() {
      Application.Run(new CheckAndRadio());
   }
}//:~
```

List, Combo, and CheckedListBoxes

Radio buttons and check boxes are appropriate for selecting among a small number of choices, but the task of selecting from larger sets of options is the work of the **ListBox**, the **ComboBox**, and the **CheckedListBox**.

This example shows the basic use of a **ListBox**. This **ListBox** allows for only a single selection to be chosen at a time; if the **SelectionMode** property is set to **MultiSimple** or **MultiExtended**, multiple items can

be chosen (**MultiExtended** should be used to allow SHIFT, CTRL, and arrow shortcuts). If the selection mode is **SelectionMode.Single**, the **Item** property contains the one-and-only selected item, for other modes the **Items** property is used.

```
//:c13:ListBoxDemo.cs
//Demonstrates ListBox selection
using System;
using System.Drawing;
using System.Windows.Forms;

class ListBoxDemo : Form{
   ListBoxDemo(){
      ListBox lb = new ListBox();
      for(int i = 0; i < 10; i++){
         lb.Items.Add(i.ToString());
      }
      lb.Location = new Point(10, 10);
      lb.SelectedValueChanged +=
         new EventHandler(OnSelect);
      Controls.Add(lb);
   }

   public void OnSelect(object src, EventArgs ea){
      ListBox lb = (ListBox) src;
      Console.WriteLine(lb.SelectedItem);
   }

   public static void Main(){
      Application.Run(new ListBoxDemo());
   }
}///:~
```

The **ComboBox** is similarly easy to use, although it can only be used for single selection. This example demonstrates the **ComboBox**, including its ability to sort its own contents:

```
//:c13:ComboBoxDemo.cs
///Demonstrates the ComboBox
using System;
using System.Drawing;
using System.Windows.Forms;
```

```
class ComboBoxDemo : Form {
    ComboBox presidents;
    CheckBox sorted;

    ComboBoxDemo() {
        ClientSize = new Size(320, 200);
        Text = "ComboBox Demo";

        presidents = new ComboBox();
        presidents.Location = new Point(10, 10);
        presidents.Items.AddRange(
          new string[]{
          "Washington", "Adams J", "Jefferson",
          "Madison", "Monroe", "Adams, JQ", "Jackson",
          "Van Buren", "Harrison", "Tyler", "Polk",
          "Taylor", "Fillmore", "Pierce", "Buchanan",
          "Lincoln", "Johnson A", "Grant", "Hayes",
          "Garfield", "Arthur", "Cleveland",
          "Harrison", "McKinley", "Roosevelt T",
          "Taft", "Wilson", "Harding", "Coolidge",
          "Hoover", "Roosevelt FD", "Truman",
          "Eisenhower", "Kennedy", "Johnson LB",
          "Nixon", "Ford", "Carter", "Reagan",
          "Bush G", "Clinton", "Bush GW"});
        presidents.SelectedIndexChanged +=
           new EventHandler(OnPresidentSelected);

        sorted = new CheckBox();
        sorted.Text = "Alphabetically sorted";
        sorted.Checked = false;
        sorted.Click +=
           new EventHandler(NonReversibleSort);
        sorted.Location = new Point(150, 10);

        Button btn = new Button();
        btn.Text = "Read selected";
        btn.Click += new EventHandler(GetPresident);
        btn.Location = new Point(150, 50);

        Controls.AddRange(
```

```
        new Control[]{presidents, sorted, btn});
    }

    private void NonReversibleSort
       (object sender, EventArgs args) {
        //bug, since non-reversible
        presidents.Sorted = sorted.Checked;
    }

    private void OnPresidentSelected
       (object sender, EventArgs args) {
        int selIdx = presidents.SelectedIndex;
        if (selIdx > -1) {
            System.Console.WriteLine(
              "Selected president is: "
              + presidents.Items[selIdx]);
        } else {
            System.Console.WriteLine(
              "No president is selected");
        }
    }

    private void GetPresident
        (object sender, EventArgs args) {
        //Doesn't work, since can be blank
        // or garbage value
        System.Console.WriteLine(presidents.Text);
        //So you have to do something like this...
        string suggestion = presidents.Text;
        if(presidents.Items.Contains(suggestion)){
            System.Console.WriteLine(
            "Selected president is: " + suggestion);
        }else{
            System.Console.WriteLine(
            "No president is selected");
        }
    }

    public static void Main() {
        Application.Run(new ComboBoxDemo());
    }
```

```
}//:~
```

After the names of the presidents are loaded into the **ComboBox**, a few handlers are defined: the check box will trigger **NonReversibleSort( )** and the button will trigger **GetPresident( )**. The implementation of **NonReversibleSort( )** sets the **ComboBox**'s **Sorted** property depending on the selection state of the **sorted Checkbox**. This is a defect as, once sorted, setting the **Sorted** property to **false** will *not* return the **ComboBox** to its original chronologically-ordered state.

GetPresident( ) reveals another quirk. The value of **ComboBox.Text** is the value of the editable field in the **ComboBox**, even if no value has been chosen, or if the user has typed in non-valid data. In order to confirm that the data in **ComboBox.Text** is valid, you have to search the **Items** collection for the text, as demonstrated.

The **CheckedListBox** is the most complex of the list-selection controls. This example lets you specify your musical tastes, printing your likes and dislikes to the Console.

```
//:c13:CheckedListBoxDemo.cs
///Demonstrates the CheckedListBox
using System;
using System.Drawing;
using System.Windows.Forms;

class CheckedListBoxDemo : Form {
   CheckedListBox musicalTastes;

   CheckedListBoxDemo(){
      ClientSize = new Size(320, 200);
      Text = "CheckedListBox Demo";

      musicalTastes = new CheckedListBox();
      musicalTastes.Location = new Point(10, 10);
      musicalTastes.Items.Add(
         "Classical", CheckState.Indeterminate);
      musicalTastes.Items.Add(
         "Jazz", CheckState.Indeterminate);
      musicalTastes.Items.AddRange(
         new string[]{
            "Blues", "Rock",
```

```
            "Punk", "Grunge", "Hip hop"});

      MakeAllIndeterminate();

      Button getTastes = new Button();
      getTastes.Location = new Point(200, 10);
      getTastes.Width = 100;
      getTastes.Text = "Get tastes";
      getTastes.Click +=
        new EventHandler(OnGetTastes);

      Controls.Add(musicalTastes);
      Controls.Add(getTastes);
   }

   private void MakeAllIndeterminate(){
      CheckedListBox.ObjectCollection items =
         musicalTasts.Items;
      for (int i = 0; i < items.Count; i++) {
         musicalTastes.SetItemCheckState(i,
           CheckState.Indeterminate);
      }
   }

   private void OnGetTastes(object o, EventArgs args){
      //Returns checked _AND_ indeterminate!
      CheckedListBox.CheckedIndexCollection
        checkedIndices = musicalTastes.CheckedIndices;
      foreach(int i in checkedIndices){
         if (musicalTastes.GetItemCheckState(i)
             != CheckState.Indeterminate) {
            System.Console.WriteLine(
               "Likes: " + musicalTastes.Items[i]);
         }
      }

      //Or, to iterate over the whole collection
      for (int i = 0;
           i < musicalTastes.Items.Count; i++) {
         if (musicalTastes.GetItemCheckState(i)
             == CheckState.Unchecked) {
```

```
            System.Console.WriteLine(
               "Dislike: " + musicalTastes.Items[i]);
         }
      }
   }
   public static void Main(){
      Application.Run(new CheckedListBoxDemo());
   }
}//:~
```

The **CheckedListBoxDemo( )** constructor shows that items can be added either one at a time, with their **CheckState** property defined, or en masse.

The **OnGetTastes( )** method shows a defect in the **CheckedListBox** control; the **CheckedIndices** property returns not just those items with **CheckState.Checked**, but also those with **CheckState.Indeterminate**! An explicit check must be added to make sure that the value at the index really *is* checked. Once an appropriate index is in hand, the value can be retrieved by using the **Items[ ]** array operator.

Multiplane displays with the **Splitter** control

It is often desirable to allow the user to change the proportion of a window devoted to various logical groups; windows with such splits are often said to be divided into multiple "panes." The **Splitter** control allows the user to resize controls manually.

We discussed the use of **GroupBox** as one way to logically group controls such as **RadioButton**s. A more general-purpose grouping control is the **Panel**, which by default is invisible. By placing logically related **Control**s on a **Panel**, and then associating a **Splitter** with the **Panel**, multipane UIs can be created easily. A **Splitter** is associated with a **Control** via the **Dock** property – the **Splitter** resizes the **Control** placed immediately *after* it in the container. The **Control** to be resized and the **Splitter** should be assigned the same **Dock** value.

In this example, we use **Panel**s that we make visible by setting their **BackColor** properties.

```
//:c13:SplitterDemo.cs
using System;
using System.Drawing;
using System.Windows.Forms;

class SplitterDemo : Form{
   SplitterDemo(){
      Panel r = new Panel();
      r.BackColor = Color.Red;
      r.Dock = DockStyle.Left;
      r.Width = 200;

      Panel g = new Panel();
      g.BackColor = Color.Green;
      g.Dock = DockStyle.Fill;

      Panel b = new Panel();
      b.BackColor = Color.Blue;
      b.Dock = DockStyle.Right;
      b.Width = 200;

      Splitter rg = new Splitter();
      //Set dock to same as resized control (p1)
      rg.Dock = DockStyle.Left;

      Splitter gb = new Splitter();
      //Set dock to same as resized control (p3)
      gb.Dock = DockStyle.Right;

      Controls.Add(g);

      //Splitter added _before_ panel
      Controls.Add(gb);
      Controls.Add(b);

      //Splitter added _before_ panel
      Controls.Add(rg);
      Controls.Add(r);
```

```
        Width = 640;
    }

    public static void Main(){
        Application.Run(new SplitterDemo());
    }
}//:~
```

After creating panels **r**, **g**, and **b** and setting their **BackColor**s appropriately, we create a **Splitter rg** with the same **Dock** value as the **r Panel** and another called **gb** with the same **Dock** value as **b**. It is critical that the **Splitter**s are added to the **Form** immediately prior to the **Panel**s they resize. The example starts with **r** and **b** at their preferred **Width** of 200 pixels, while the entire **Form** is set to take 640. However, you can resize the **Panel**s manually.

TreeView & ListView

Everyone seems to have a different idea of what the ideal **TreeView** control should look like and every discussion group for every UI toolkit is regularly swamped with the intricacies of **TreeView** programming. Windows Forms is no exception, but the general ease of programming Windows Forms makes basic **TreeView** programming fairly straightforward. The core concept of the **TreeView** is that a **TreeView** contains **TreeNodes**, which in turn contain other **TreeNodes**. This model is essentially the same as the Windows Forms model in which **Control**s contain other **Control**s. So just as you start with a **Form** and add **Control**s and **Control**s to those **Control**s, so too you create a **TreeView** and add **TreeNode**s and **TreeNode**s to those **TreeNode**s.

However, the general programming model for Windows Forms is that events are associated with the pieces that make up the whole (the **Control**s within their containers), while the programming model for the **TreeView** is that events are associated with the whole; **TreeNode**s have no events.

This example shows the simplest possible use of **TreeView**.

```
//:c13:TreeViewDemo1.cs
//Demonstrates TreeView control
using System;
using System.Drawing;
using System.Windows.Forms;

class TreeViewDemo1 : Form{
   TreeViewDemo1(){
      TreeView tv = new TreeView();
      TreeNode root = new TreeNode("Fish");
      TreeNode cart = new TreeNode("Sharks & Rays");
      TreeNode bony = new TreeNode("Bony fishes");
      tv.Nodes.Add(root);
      root.Nodes.Add(cart);
      root.Nodes.Add(bony);

      tv.AfterSelect +=
         new TreeViewEventHandler(AfterSelectHandler);
      Controls.Add(tv);
   }

   public void AfterSelectHandler
      (object src, TreeViewEventArgs a){
      TreeNode sel = ((TreeView) src).SelectedNode;
      System.Console.WriteLine(sel);
   }
   public static void Main(){
      Application.Run(new TreeViewDemo1());
   }
}//:~
```

In the constructor, after a **TreeView** control is initialized, three **TreeNode**s are created. The **root** node is added to the **Nodes** collection of the **TreeView**. The branch nodes **cart** and **bony** are added to the **Nodes** collection, not of the **TreeView**, but of the **root** node. A **TreeViewEventHandler** delegate is created to output the value of the selected node to the Console. The delegate is added to the **TreeView**'s **AfterSelect** event – the **TreeView** has a dozen unique events relating to node selection, collapsing and expanding, and changes to the item's checkbox (itself an optional property).

ListView

ListView may finally replace **TreeView** as "most discussed UI widget." The **ListView** is an insanely customizable widget that is similar to the right-hand side of Windows Explorer – items placed within a **ListView** can be viewed in a detail view, as a list, and as grids of large or small icons. Like the **TreeView**, the **ListView** uses a basic containment model: a **TreeView** contains a collection of **ListViewItem**s which do not themselves have events. The **ListViewItem**s are added to the **Items** property of the **ListView**. Various properties of the **ListView** and the individual **ListViewItem**s control the various displays.

Icon Views

The **ListView** contains two **ImageList**s, which hold icons (or other small graphics) that can be associated with different types of **ListViewItem**s. One **ImageList** should be assigned to the **SmallImageList** and the other to the **LargeImageList** property of the **ListView**. Corresponding images should be added to both **ImageList**'s at the same offsets, since the selection of *either* **ImageList** is determined by the **ListViewItem**'s **ImageIndex** property. In other words, if the **ListViewItem.ImageIndex** is set to 3, if **ListView.View** is set to **ListView.LargeIcon** the 4th image in **LargeImageList** will be displayed for that **ListViewItem**, while if **ListView.View == View.SmallIcon**, the 4th image in **SmallImageList** will be displayed.

Details View

The details view of a **ListView** consists of a series of columns, the first of which displays the **ListViewItem** and its associated icon from the **SmallImageList**. Subsequent columns display text describing various aspects related to the **ListViewItem**. The text displayed by the 2nd and subsequent columns is determined by the collection of **ListViewSubItem**s in the **ListViewItem**'s **SubItems** property. The column header text is set with the **ListView.Columns** property. The programmer is responsible for coordinating the consistency of column header offsets and the indices of **ListViewSubItem**s.

This example demonstrates the **ListView**'s various modes.

```
//:c13:ListViewDemo.cs
//Demonstrates the ListView control
using System;
using System.IO;
using System.Drawing;
using System.Windows.Forms;

class ListViewDemo : Form{
   ListView lv;

   ListViewDemo(){
      //Set up control panel
      Panel p = new Panel();
      RadioButton[] btn = new RadioButton[]{
         new RadioButton(), new RadioButton(),
         new RadioButton(), new RadioButton()
      };
      btn[0].Checked = true;
      btn[0].Text = "Details";
      btn[1].Text = "Large Icons";
      btn[2].Text = "List";
      btn[3].Text = "Small Icons";
      for(int i = 0; i < 4; i++){
         btn[i].Location =
            new Point(10, 20 * (i + 1));
         btn[i].CheckedChanged +=
            new EventHandler(SelectView);
      }
      p.Controls.AddRange(btn);
      p.Dock = DockStyle.Left;
      p.Width = 100;

      Splitter s = new Splitter();
      s.Dock = DockStyle.Left;

      //ListView initial stuff
      lv = new ListView();
      lv.Dock = DockStyle.Fill;
```

```
lv.Columns.Add(
   "File Name", 150, HorizontalAlignment.Left);
lv.Columns.Add(
   "Size", 150, HorizontalAlignment.Left);
lv.View = View.Details;

//Load images
Image smallCS = Image.FromFile("cs_sm.bmp");
Image smallExe = Image.FromFile("exe_sm.bmp");
ImageList il = new ImageList();
il.Images.Add(smallCS);
il.Images.Add(smallExe);
lv.SmallImageList = il;

Image largeCS = Image.FromFile("cs_lrg.bmp");
Image largeExe = Image.FromFile("exe_lrg.bmp");
ImageList il2 = new ImageList();
il2.Images.Add(largeCS);
il2.Images.Add(largeExe);
lv.LargeImageList = il2;

DirectoryInfo dir = new DirectoryInfo(".");
foreach(FileInfo f in dir.GetFiles("*.*")){
   string fName = f.Name;
   string size = f.Length.ToString();
   string ext = f.Extension;

   ListViewItem item = new ListViewItem(fName);
   if(ext == ".cs"){
      item.ImageIndex = 0;
   }else{
      item.ImageIndex = 1;
   }
   item.SubItems.Add(size);
   lv.Items.Add(item);
}

Controls.Add(lv);
Controls.Add(s);
Controls.Add(p);
Width = 640;
```

```
    }

    public void SelectView(Object src, EventArgs ea){
       RadioButton rb = (RadioButton) src;
       string viewDesired = rb.Text;
       switch(viewDesired){
       case "Details" :
          lv.View = View.Details;
          break;
       case "Large Icons" :
          lv.View = View.LargeIcon;
          break;
       case "List" :
          lv.View = View.List;
          break;
       case "Small Icons" :
          lv.View = View.SmallIcon;
          break;
       }
    }

    public static void Main(){
       Application.Run(new ListViewDemo());
    }
}//:~
```

The **ListViewDemo( )** constructor defines a **Panel** and **RadioButton** array to select the various **ListView.View** values. A **Splitter** is also defined to allow this panel to be resized. The **ListView lv** is created and its **Dock** property set to **DockStyle.Fill**. We specify that when the **ListView** is switched to details view, the two columns will be labeled "File Name" and "Size," each will initially be 150 pixels wide, and each will display its text left-aligned. We then set **lv**'s initial view state to, in fact, be **View.Details**.

The next section of the constructor, labeled "LoadImages," loads small and then larger images, and places the corresponding images (**cs_sm.bmp** and **cs_lrg.bmp** and **exe_sm.bmp** and **exe_lrg.bmp**) in **ImageList**s which are assigned to the **ListView**'s **SmallImageList** and **LargeImageList** properties.

The **ListView** is populated with the files in the current directory. For each file found, we determine the name, file length, and extension and create a **ListViewItem** with it's "main" **Text** set to the file's name. If the extension is ".cs" we set that **ListViewItem**'s **IconIndex** to correspond to the C# images in the **ListView**'s **ImageList**s, otherwise we set the index to 1.

The **string** value representing the file's length is added to the **SubItems** collection of the **ListViewItem**. The **ListViewItem** itself is always the *first* item in the **SubItems** list; in this case that will appear in the "File Name" column of the **TreeView**. The **size** string will appear in the *second* column of the **TreeView**, under the "Size" heading.

The last portion of the constructor adds the **Control**s to the **ListViewDemo** form in the appropriate order (remember, because there's a **Splitter** involved, the **Splitter** must be added to the form immediately before the **Panel** it resizes).

The **SelectView( )** delegate method, called by the **SelectionChanged** event of the **RadioButton**s, sets **ListView.View** to the appropriate value from the **View** enumeration.

Clipboard and Drag-and-Drop

Another perennially challenging interface issue is using the Clipboard and supporting drag-and-drop operations.

Clipboard

Objects on the Clipboard are supposed to be able to transform themselves into a variety of formats, depending on what the pasting application wants. For instance, a vector drawing placed on the Clipboard should be able to transform itself into a bitmap for Paint or a Scalable Vector Graphics document for an XML editor. Data transfer in Windows Forms is mediated by classes which implement the **IDataObject** interface. A consuming application or **Control** first gets a reference to the **Clipboard**'s current **IDataObject** by calling the static method

Clipboard.GetDataObject( ). Once in hand, the **IDataObject** method **GetDataPresent( )** is called with the desired type in a **string** or **Type** argument. If the **IDataObject** can transform itself into the requested type, **GetDataPresent( )** returns **true**, and **GetData( )** can then be used to retrieve the data in the requested form.

In this example, one **Button** places a **string** on the Clipboard, and the other button puts the Clipboard's content (if available as a **string**) into a **Label**. If you copy an image or other media that cannot be transformed into a **string** onto the Clipboard and attempt to paste it with this program, the label will display "Nothing."

```
//:c13:ClipboardDemo.cs
//Cuts to and pastes from the system Clipboard
using System;
using System.Drawing;
using System.Threading;
using System.Windows.Forms;

class ClipboardDemo : Form {
    Label l;

    ClipboardDemo(){
      Button b = new Button();
      b.Location = new Point(10, 10);
      b.Text = "Clip";
      b.Click +=
        new EventHandler(AddToClipboard);
      Controls.Add(b);

      Button b2 = new Button();
      b2.Location = new Point(100, 10);
      b2.Text = "Paste";
      b2.Click +=
          new EventHandler(CopyFromClip);
      Controls.Add(b2);

      l = new Label();
      l.Text = "Nothing";
      l.Location = new Point(100, 50);
      l.Size = new Size(200, 20);
```

```
      Controls.Add(l);
   }

   public void AddToClipboard(Object s, EventArgs a){
      Clipboard.SetDataObject("Text. On clipboard.");
   }

   public void CopyFromClip(Object s, EventArgs a){
      IDataObject o = Clipboard.GetDataObject();
      if(o.GetDataPresent(typeof(string))){
        l.Text = o.GetData(typeof(string)).ToString();
      }else{
        l.Text = "Nothing";
      }
   }

   public static void Main(){
      Application.Run(new ClipboardDemo());
   }
}//:~
```

Nothing unusual is done in the **ClipboardDemo( )** constructor; the only thing somewhat different from other demos is that the **Label l** is made an instance variable so that we can set its contents in the **CopyFromClip( )** method.

The **AddToClipboard( )** method takes an **object**, in this case a **string**. If the Clipboard is given a **string** or **Image**, that value can be pasted into other applications (use **RichTextBox.Rtf** or **SelectedRtf** for pasting into Microsoft Word).

CopyFromClip( ) must be a little cautious, as there is no guarantee that the Clipboard contains data that can be turned into a **string** (every .NET Framework **object** can be turned into a **string**, but the Clipboard may very well contain data that is not a .NET Framework **object**). If the **IDataObject** instance can indeed be expressed as a **string**, we retrieve it, again specifying **string** as the type we're looking for. Even though we've specified the type in the argument, we still need to change the return value of **GetData( )** from **object** to **string**, which we do by calling the **ToString( )** method (we could also have used a **(string)** cast).

Drag and Drop

Drag and drop is a highly visual form of data transfer between components and applications. In Windows Forms, drag and drop again uses the **IDataObject** interface, but also involves visual feedback. In this example, we demonstrate both drag-and-drop and the creation of a custom **IDataObject** that can transform itself into a variety of types.

An exemplar of something that transforms into a variety of types is, of course, SuperGlo, the floorwax that tastes like a dessert topping (or is it the dessert topping that cleans like a floorwax? All we know for sure is that it's delicious *and* practical!)

```
//:c13:SuperGlo.cs
//Compile with:
//csc DragAndDropDemo.cs SuperGlo.cs SuperGloMover.cs
//Domain objects for custom drag-and-drop

interface FloorWax{
    void Polish();
}

interface DessertTopping{
    void Eat();
}

class SuperGlo : FloorWax, DessertTopping {
    public void Polish() {
        System.Console.WriteLine(
          "SuperGlo makes your floor shine!");
    }
    public void Eat() {
        System.Console.WriteLine(
          "SuperGlo tastes great!");
    }
}//:~ (example continues with SuperGloMover.cs)
```

In this listing, we define the domain elements – a **FloorWax** interface with a **Polish( )** method, a **DessertTopping** interface with an **Eat( )** method, and the **SuperGlo** class, which implements both the interfaces

by writing to the Console. Now we need to define an **IDataObject** implementation that is versatile enough to demonstrate **SuperGlo**:

```
//(continuation)
//:c13:SuperGloMover.cs
//Custom IDataObject, can expose SuperGlo as:
//FloorWax, DessertTopping, or text

class SuperGloMover : IDataObject {
    //Concern 1: What formats are supported?
    public string[] GetFormats() {
        return new string[]{
            "FloorWax", "DessertTopping", "SuperGlo"};
    }

    public string[] GetFormats(bool autoConvert) {
        if (autoConvert) {
            return new string[]{
                "FloorWax", "DessertTopping",
                "SuperGlo",  DataFormats.Text};
        } else {
            return GetFormats();
        }
    }

   //Concern 2: Setting the data

   //Storage
   SuperGlo superglo;

   public void SetData(object o) {
        if (o is SuperGlo) {
            superglo = (SuperGlo) o;
        }
   }

   public void SetData(string fmt, object o) {
      if (fmt == "FloorWax" || fmt == "DessertTopping"
            || fmt == "SuperGlo") {
            SetData(o);
      } else {
```

```
        if (fmt == DataFormats.Text) {
            superglo = new SuperGlo();
        } else {
            System.Console.WriteLine(
              "Can't set data to type " + fmt);
        }
    }
}

public void SetData(Type t, object o) {
    SetData(t.Name, o);
}

public void SetData
  (String fmt, bool convert, object o) {
   if (fmt == DataFormats.Text
        && convert == false) {
        System.Console.WriteLine(
          "Refusing to change a string " +
          "to a superglo");
    } else {
        SetData(fmt, o);
    }
}

//Concern 3: Is there a format client can use?
public bool GetDataPresent(string fmt) {
  if (fmt == "DessertTopping" || fmt == "FloorWax"
        || fmt == DataFormats.Text
        || fmt == "SuperGlo") {
        return true;
  } else {
        return false;
  }
}

public bool GetDataPresent(Type t) {
    return(GetDataPresent(t.Name));
}

public bool GetDataPresent
```

```
(String fmt, bool convert) {
    if (fmt == DataFormats.Text
        && convert == false) {
        return false;
    } else {
        return GetDataPresent(fmt);
    }
}

//Concern 4: Get the data in requested format
public object GetData(string fmt) {
    switch (fmt) {
    case "FloorWax" :
      return superglo;
    case "DessertTopping" :
      return superglo;
    case "SuperGlo" :
      return superglo;
    case "Text" :
      return "SuperGlo -- It's a FloorWax! "
      + "And a dessert topping!";
    default :
      Console.WriteLine(
        "SuperGlo is many things, but not a "
        + fmt);
      return null;
    }
}

public object GetData(Type t) {
    string fmt = t.Name;
    return GetData(fmt);
}

public object GetData(string fmt, bool convert) {
    if (fmt == DataFormats.Text
        && convert == false) {
        return null;
    } else {
        return GetData(fmt);
    }
```

```
    }
}//:~ (example continues with DragAndDropDemo.cs)
```

To implement **IDataObject**, you must address 4 concerns: what are the formats that are supported, setting the data, is the data in a format the client can use, and getting the data in the specific format the client wants.

The first concern is addressed by the two overloaded **GetFormats( )** methods. Both return **string** arrays which represent the .NET classes into which the stored data can be transformed. In addition to the types which the **SuperGlo** class actually instantiates, we also specify that **SuperGlo** can automatically be converted to-and-from text. The **DataFormats** class contains almost two dozen static properties defining various Clipboard formats that Windows Forms already understands.

Setting the data is done, in the simplest case, by passing in an **object**, which is stored in the instance variable **superglo**. If a person attempts to store a non-**SuperGlo** object, the request will be quietly ignored. The .NET documentation is silent on whether **SetData( )** should throw an exception if called with an argument of the wrong type; and typically silence means that one should *not* throw an exception. This goes against the grain of good programming practice (usually, the argument to a method should always either affect the return value of the method, affect the state of the object or the state of the argument, or result in an exception).

The second **SetData( )** method takes a **string** representing a data format and an object. If the **string** is any one of the native types of **SuperGlo** ("SuperGlo", "Dessert", or "FloorWax"), the object is passed to the **SetData(object)** method. If the **string** is set to **DataFormats.Text**, a new **SuperGlo** is instantiated and stored. If the **string** is not one of these four values, a diagnostic is printed and the method fails. The third **SetData( )** method takes a **Type** as its first argument and simply passes that argument's **Name** property forward to **SetData(string, object)**.

The fourth-and-final **SetData( )** method takes, in addition to a format **string** and the data **object** itself, a **bool** that may be used to turn off auto-conversion (in this case, the "conversion" from **DataFormat.Text** that just instantiates a new **SuperGlo**).

The **GetDataPresent( )** methods return **true** if the argument is one of the **SuperGlo** types or **DataTypes.Text** (except if the **convert** argument is set to **false**).

The **GetData( )** method returns a reference to the stored **SuperGlo** if the format requested is "SuperGlo", "FloorWax", or "DessertTopping." If the format requested is "Text," the method returns a promotional reference. If anything else, **GetData( )** returns a **null** reference.

Now that we have the domain objects and **SuperGloMover**, we can put it all together visually:

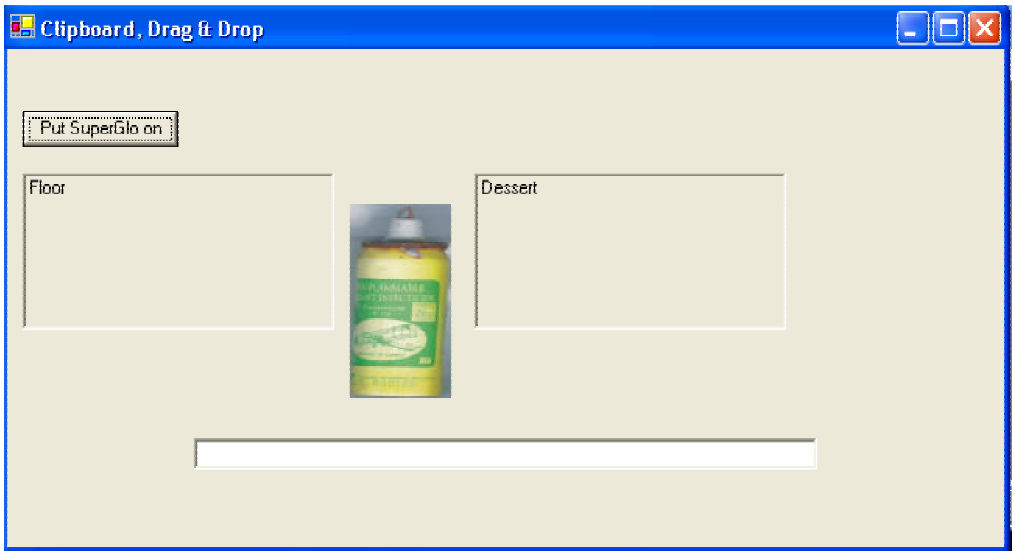
```
//(continuation)
//:c13:DragAndDropDemo.cs
//Demonstrates drag-and-drop with SuperGlo and
//SuperGloMover
using System;
using System.Drawing;
using System.Windows.Forms;

class DragAndDropDemo : Form {
    DragAndDropDemo() {
        ClientSize = new Size(640, 320);
        Text = "Drag & Drop";

        Button b = new Button();
        b.Text = "Put SuperGlo on clipboard";
        b.Location = new Point(10, 40);
        b.Width = 100;
        b.Click += new EventHandler(OnPutSuperGlo);

        PictureBox pb = new PictureBox();
        pb.Image = Image.FromFile(@".\superglo.jpg");
        pb.SizeMode = PictureBoxSizeMode.AutoSize;
        pb.Location = new Point(220, 100);
        pb.MouseDown +=
          new MouseEventHandler(OnBeginDrag);

        Panel floor = new Panel();
        floor.BorderStyle = BorderStyle.Fixed3D;
        floor.Location = new Point(10, 80);
```

```
        Label f = new Label();
        f.Text = "Floor";
        floor.Controls.Add(f);
        floor.AllowDrop = true;
        floor.DragEnter +=
          new DragEventHandler(OnDragEnterFloorWax);
        floor.DragDrop +=
          new DragEventHandler(OnDropFloorWax);

        Panel dessert = new Panel();
        dessert.BorderStyle = BorderStyle.Fixed3D;
        dessert.Location = new Point(300, 80);
        Label d = new Label();
        d.Text = "Dessert";
        dessert.Controls.Add(d);
        dessert.AllowDrop = true;
        dessert.DragEnter +=
          new DragEventHandler(
            OnDragEnterDessertTopping);
        dessert.DragDrop +=
          new DragEventHandler(OnDropDessertTopping);


        TextBox textTarget = new TextBox();
        textTarget.Width = 400;
        textTarget.Location = new Point(120, 250);
        textTarget.AllowDrop = true;
        textTarget.DragEnter +=
        new DragEventHandler(OnDragEnterText);
        textTarget.DragDrop +=
        new DragEventHandler(OnDropText);

        Controls.AddRange(
            new Control[]{
              b, pb, floor, dessert, textTarget});
    }

    private void OnPutSuperGlo(object s, EventArgs a){
        SuperGlo superglo = new SuperGlo();
        SuperGloMover mover = new SuperGloMover();
        mover.SetData(superglo);
```

```
        Clipboard.SetDataObject(mover);
    }

    private void OnBeginDrag
    (object s, MouseEventArgs args) {
        SuperGloMover sgm = new SuperGloMover();
        sgm.SetData(new SuperGlo());
        ((Control) s).DoDragDrop(
            sgm, DragDropEffects.Copy );
    }

    private void OnDragEnterFloorWax
    (object s, DragEventArgs args) {
        if (args.Data.GetDataPresent("FloorWax")) {
            args.Effect = DragDropEffects.Copy;
        }
    }

    private void OnDropFloorWax
    (object s, DragEventArgs args) {
        FloorWax f =
        (FloorWax) args.Data.GetData("FloorWax");
        f.Polish();
    }

    private void OnDragEnterDessertTopping
    (object s, DragEventArgs args) {
       if(args.Data.GetDataPresent("DessertTopping"))
{
            args.Effect = DragDropEffects.Copy;
        }
    }

    private void OnDropDessertTopping
      (object s, DragEventArgs args) {
        DessertTopping d =
        (DessertTopping) args.Data.GetData
        ("DessertTopping");
        d.Eat();
    }
```

```
    private void OnDragEnterText
    (object s, DragEventArgs args) {
        if (args.Data.GetDataPresent("Text")) {
            args.Effect = DragDropEffects.Copy;
        }
    }

    private void OnDropText
    (object sender, DragEventArgs args) {
        string s = (string) args.Data.GetData("Text");
        ((Control)sender).Text = s;
    }

    public static void Main() {
        Application.Run(new DragAndDropDemo());
    }
}
//:~
```

When run, this application looks like this:



If you click on the image, you can drag it to either of the two **Panel**s, the **TextEdit** display, or into a text editor such as Microsoft Word. As you drag, the mouse cursor will change to indicate whether or not the

SuperGlo can be dropped. If you drop the **SuperGlo** on the "Floor" panel you'll see one message on the console, if on the "Dessert" panel another, and if in the **TextEdit** box or in another application, the promotional message will be pasted. If you click on the button, you can put a new **SuperGlo** on the Clipboard.

The **DragAndDropDemo( )** constructor should have no surprises until we add a **MouseEventHandler** on for the **MouseDown** event of the **PictureBox** control. That event will trigger the creation of a new **SuperGloMover** (as will the pressing of the **Button).**

The **Panel**s both have their **AllowDrop** property set to **true**, so when the mouse enters them while dragging an object, the mouse will give visual feedback that a drop target is being traversed. Additionally, the **Panel**'s **DragEnter** and **DragDrop** events are wired to event handlers.

When the button is pressed, it ends up calling **OnPutSuperGlo( )**. A new **SuperGlo** domain object is created, as is a new **SuperGloMover**. The **mover**'s data is set to the just-created **SuperGlo** and the **mover** is placed on the system Clipboard. In another application such as a text editor that accepts Clipboard text data, you should be able to paste data. That other application will end up calling **SuperGloMover.GetData( )** requesting **DataType.Text**, resulting in the message "SuperGlo – It's a FloorWax! And a dessert topping!" being pasted into the other application.

The method **OnBeginDrag( )** is pretty similar to **OnPutSuperGlo( )** but after the **SuperGloMover** is instantiated, the method **DoDragDrop( )** is called on the *originating* **Control**. This is somewhat confusing, as you might think that the destination should be in charge, but on reflection it makes sense that the originator knows the most about the data, even when the data is "far away" on the screen. There are several different **DragDropEffects**, which are bitwise combinable; in this case, the effect we want is to copy the data to the target.

Both **Panel**s have very similar **OnDragEnter*xxx*( )** methods. The **DragEventArgs args** has a **Data** property that contains the **SuperGloMover** created in **OnBeginDrag( ).** This **Data** is checked to make sure that it can present the appropriate type ("FloorWax" or

"DessertTopping") and if so, setting the **args.Effect** property makes the mouse show the appropriate visual cue for **DragDropEffects.Copy**.

Similarly, both **Panel**s have similar **OnDrop*xxx*( )** methods. The **Data** property (the **SuperGloMover**) has its **GetData( )** method called with the desired type, either a **FloorWax** or **DessertTopping**. This will return the **SuperGlo**. Either **FloorWax.Polish( )** or **DessertTopping.Eat( )** is called, and the appropriate message printed on the Console.

The **TextBox** has similar **OnDragEnterText( )** and **OnDropText( )** methods, except the **SuperGloMover** is queried for the "Text" type.

Data-bound Controls

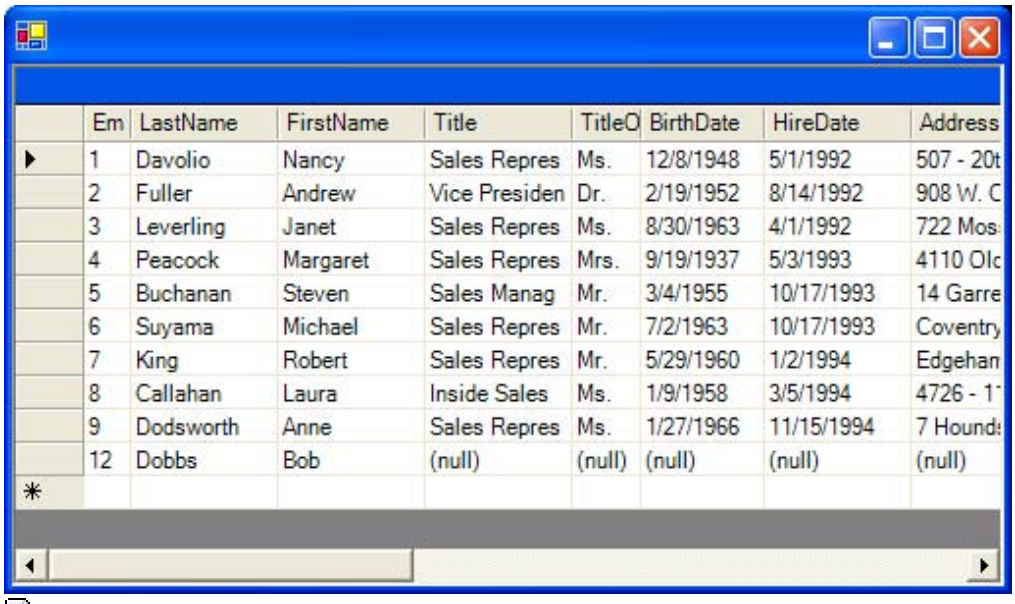
Windows Forms and ADO.NET combine to make data-driven user interfaces very easy to program. In Chapter #ref#, we saw how ADO.NET separates the concerns of moving data in and out from a persistent datastore from the concerns of manipulating the in-memory **DataSet**. Once a **DataSet** is populated, instances of the **DataBinding** class mediate between **Control**s and **DataSet**s.

One of the more impressive **Control**s for displaying data is the **DataGrid**, which can display all the columns in a table. This example revisits the "Northwind.mdb" Access database:

```
//:c13:DataBoundDemo.cs
//Demonstrates the basics of data-binding
using System;
using System.Data;
using System.Data.OleDb;
using System.Windows.Forms;

class DataBoundDemo : Form {
   OleDbDataAdapter adapter;
   DataSet emps;

   DataBoundDemo(){
      DataGrid dg = new DataGrid();
      dg.Dock = DockStyle.Fill;
```

```
      Controls.Add(dg);

      ReadEmployees("NWind.mdb");

      dg.SetDataBinding(emps, "Table");
   }

   private void ReadEmployees(string pathToAccessDB){
      OleDbConnection cnctn = new OleDbConnection();
      cnctn.ConnectionString=
      "Provider=Microsoft.JET.OLEDB.4.0;" +
      "data source=" + pathToAccessDB;
      cnctn.Open();

      string selStr = "SELECT * FROM EMPLOYEES";
      adapter = new OleDbDataAdapter(selStr, cnctn);
      new OleDbCommandBuilder(adapter);

      emps = new DataSet("Employees");
      adapter.Fill(emps);
   }
   public static void Main(){
      Application.Run(new DataBoundDemo());
   }
}//:~
```

A **DataGrid dg** control is created and set so that it will fill the entire client area of the **DataBoundDemo** form. Then, the **ReadEmployees( )** method fills the **DataSet emps**, just as in the examples in Chapter #ref#. Once **emps** is filled, the **dg.SetDataBinding( )** associates the **DataGrid** with the "Table" table in **emps**, which in this case is the only table in the **DataSet**. The result shows the contents of the Northwind database:



Em	LastName	FirstName	Title	TitleO	BirthDate	HireDate	Address
1	Davolio	Nancy	Sales Repres	Ms.	12/8/1948	5/1/1992	507 - 20t
2	Fuller	Andrew	Vice Presiden	Dr.	2/19/1952	8/14/1992	908 W. C
3	Leverling	Janet	Sales Repres	Ms.	8/30/1963	4/1/1992	722 Mos
4	Peacock	Margaret	Sales Repres	Mrs.	9/19/1937	5/3/1993	4110 Olc
5	Buchanan	Steven	Sales Manag	Mr.	3/4/1955	10/17/1993	14 Garre
6	Suyama	Michael	Sales Repres	Mr.	7/2/1963	10/17/1993	Coventry
7	King	Robert	Sales Repres	Mr.	5/29/1960	1/2/1994	Edgehan
8	Callahan	Laura	Inside Sales	Ms.	1/9/1958	3/5/1994	4726 - 1
9	Dodsworth	Anne	Sales Repres	Ms.	1/27/1966	11/15/1994	7 Hound
12	Dobbs	Bob	(null)	(null)	(null)	(null)	(null)

Even more powerfully, the **BindingManagerBase** class can coordinate a set of **Binding**s, allowing you to have several **Control**s whose databound properties are simultaneously and transparently changed to correspond to a single record in the **DataSet**. This example allows you to navigate through the employee data:

```
//:c13:Corresponder.cs
//Demonstrates how bound controls can be made to match

using System;
using System.Drawing;
using System.Windows.Forms;
using System.Data;
using System.Data.OleDb;

class Corresponder : Form {

   OleDbDataAdapter adapter;
   DataSet emps;

   Corresponder(){
      ReadEmployees("NWind.mdb");
```

```
      Label fName = new Label();
      fName.Location = new Point(10, 10);
      Binding fBound =
         new Binding("Text", emps, "Table.FirstName");
      fName.DataBindings.Add(fBound);
      Controls.Add(fName);

      Label lName = new Label();
      //! lName.DataBindings.Add(fBound);
      lName.Location = new Point(10, 40);
      Binding lBound =
         new Binding("Text", emps, "Table.LastName");
      lName.DataBindings.Add(lBound);
      Controls.Add(lName);

      Button next = new Button();
      next.Location = new Point(100, 70);
      next.Text = ">";
      next.Click += new EventHandler(OnNext);
      Controls.Add(next);

      Button prev = new Button();
      prev.Location = new Point(10, 70);
      prev.Text = "<";
      prev.Click += new EventHandler(OnPrev);
      Controls.Add(prev);
   }

   void OnNext(object src, EventArgs ea){
      BindingManagerBase mgr =
         BindingContext[emps, "Table"];
      mgr.Position++;
   }

   void OnPrev(object src, EventArgs ea){
      BindingManagerBase mgr =
         BindingContext[emps, "Table"];
      mgr.Position--;
   }
```

```
   private void ReadEmployees(
                              string pathToAccessDB){
      OleDbConnection cnctn = new OleDbConnection();
      cnctn.ConnectionString=
      "Provider=Microsoft.JET.OLEDB.4.0;" +
      "data source=" + pathToAccessDB;
      cnctn.Open();

      string selStr = "SELECT * FROM EMPLOYEES";
      adapter = new OleDbDataAdapter(selStr, cnctn);
      new OleDbCommandBuilder(adapter);

      emps = new DataSet("Employees");
      adapter.Fill(emps);
   }
   public static void Main(){
      Application.Run(new Corresponder());
   }
}//:~
```

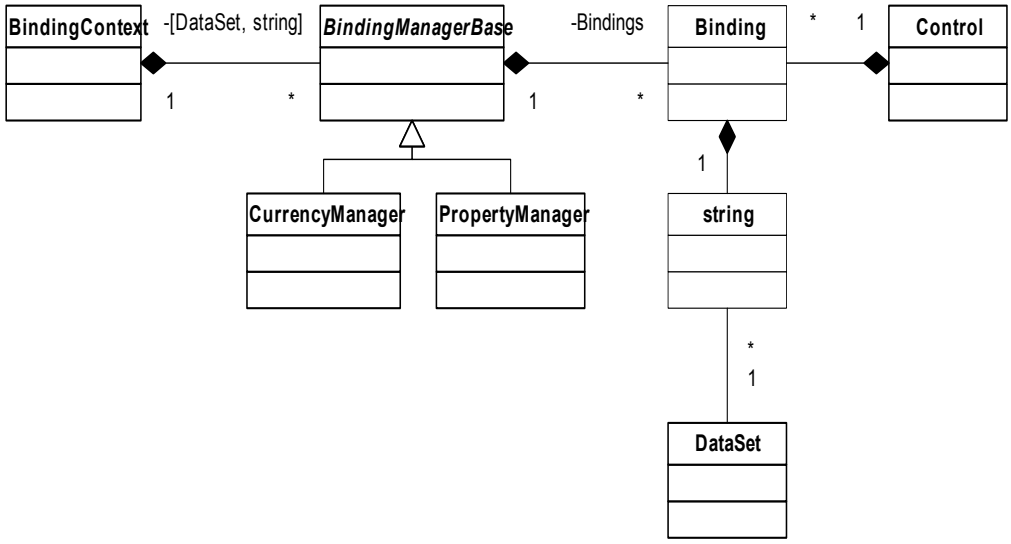
After populating the **DataSet emps** from the database, we create a **Label** to hold the first name of the current employee. A new **Binding** object is created; the first parameter specifies the name of the Property to which the data will be bound (often, this will be the "Text" property) , the second the **DataSet** that will be the data source, and the third the specific column in the **DataSet** to bind to. Once the **Binding** is created, adding it to **fName.Bindings** sets the **Label**'s **Text** property to the **DataSet** value.

The commented-out line immediately after **lName**'s constructor call is a call that will lead to a runtime exception – **Binding**s cannot be shared between **Control**s. If you wish multiple controls to reflect the same **DataSet** value, you have to create multiple **Binding**s, one for each **Control**.

A similar process is used to configure the **lName** label for the last name, and two **Button**s are created to provide rudimentary navigation.

The **Button**s' event handlers use the **BindingContext**'s static **[ ]** operator overload which takes a **DataSet** and a **string** representing a data member. The data member in this case is the "Table" result. The

resulting **BindingManagerBase.Property** value is then incremented or decremented within the event handler. As that happens, the **BindingManagerBase** updates its associated **Binding**s, which in turn update their bound **Control**s with the appropriate data from the **Binding**s' **DataSet.** This diagram illustrates the static structure of the relationships.



You'll notice that the **BindingManagerBase** is an abstract class instantiated by **CurrencyManager** and **PropertyManager**. If the call to **BindingContext[object, string]** returns an **object** that implements **IList**, you'll get a **CurrencyManager** whose **Position** property moves back and forth between items in the **IList** (this is what was shown in the example; manipulating the **Position** in the event handler switches between employees). If the call to **BindingContext[object, string]** does not return an object implementing **IList**, a **PropertyManager** is returned.

Editing Data from Bound Controls

Objects of the **Binding** class manipulate the **DataSet**. As discussed in Chapter #ADO.NET#, changing the **DataSet** does not affect the backing

store, the **IDataAdapter** is responsible for mediating between the in-memory **DataStore** and the **IDbConnection**.

This example demonstrates updating the database, provides a more detailed look at the events that occur during data binding, and illustrates a useful UI technigue for manipulating databases or other large data structures. The UI technique is based on the premise that it may be expensive in terms of memory or resources or redrawing to have every editable portion of the screen actually *be* editable; rather, the **Control** that manages the editing process is dynamically positioned on the screen at the insertion point (or, in this case, when the mouse clicks on a data-bound **Label** which we wish to edit). This technique is not worth the effort on simple screens, but really comes into its own on complex UIs such as you might have on a word processor, spreadsheet, or a workflow application. If you use this technique, you should implement it using the PAC GUI architecture so that the result appears as a seamless component.

```
//:c13:FormEdit.cs
//Demonstrates data write from Form
using System;
using System.Collections;
using System.Data;
using System.Data.OleDb;
using System.Drawing;
using System.Windows.Forms;

class FormEdit : Form{
   Hashtable colForLabel = new Hashtable();
   bool editWindowActive = false;
   TextBox editWindow = new TextBox();
   Label labelBeingEdited = null;

   OleDbDataAdapter adapter;
   DataSet emps;

   FormEdit(){
      ReadEmployees("NWind.mdb");
      InitDataStructure();
      InitLabels();
```

```
      InitCommitter();
   }

   private void InitDataStructure(){
      colForLabel[new Label()] = "FirstName";
      colForLabel[new Label()] = "LastName";
   }

   private void InitLabels(){
      int x = 10;
      int y = 10;
      int yIncrement = 35;

      foreach(string colName in colForLabel.Values){
         //Bihashtable w KeyForValue() not in library
         foreach(Label lbl in colForLabel.Keys){
            string aColName =
               (string) colForLabel[lbl];
            if (aColName == colName) {
               //Right key (label) for value (colName)
               InitLabel(lbl, colName, x, y);
               y += yIncrement;
            }
         }
      }
   }

   private void InitLabel(Label lbl, string colName,
                          int x, int y){
      lbl.Location = new Point(x, y);
      lbl.Click += new EventHandler(OnLabelClick);
      lbl.TextChanged +=
         new EventHandler(OnTextChange);
      Controls.Add(lbl);

      string navPath = "Table." + colName;
      Binding b =
      new Binding("Text", emps, navPath);
      b.Parse +=
         new ConvertEventHandler(OnBoundDataChange);
      lbl.DataBindings.Add(b);
```

```
    }

    private void InitCommitter(){
       Button b = new Button();
       b.Width = 120;
       b.Location = new Point(150, 10);
       b.Text = "Commit changes";
       b.Click += new EventHandler(OnCommit);
       Controls.Add(b);
    }

    public void OnLabelClick(Object src, EventArgs ea){
       Label srcLabel = (Label) src;
       if(editWindowActive)
          FinalizeEdit();
       PlaceEditWindowOverLabel(srcLabel);
       AssociateEditorWithLabel(srcLabel);
    }

    private void PlaceEditWindowOverLabel(Label lbl){
       editWindow.Location = lbl.Location;
       editWindow.Size = lbl.Size;
       if(Controls.Contains(editWindow) == false){
          Controls.Add(editWindow);
       }
       editWindow.Visible = true;
       editWindow.BringToFront();
       editWindow.Focus();
       editWindowActive = true;
    }

    private void AssociateEditorWithLabel(Label l){
      editWindow.Text = l.Text;
      labelBeingEdited = l;
    }

    public void FinalizeEdit(){
         System.Console.WriteLine("Finalizing edit");
         labelBeingEdited.Text = editWindow.Text;
         System.Console.WriteLine("Text changed");
         //Needed to trigger binding event
```

```
      labelBeingEdited.Focus();

      editWindow.Visible = false;
      editWindow.SendToBack();
      editWindowActive = false;
   }

   public void OnTextChange(Object src, EventArgs ea){
      Label lbl = (Label) src;
      string colName = (string) colForLabel[lbl];
      System.Console.WriteLine(
        colName + " has changed");
   }

   public void OnBoundDataChange
    (Object src, ConvertEventArgs ea){
      System.Console.WriteLine("Bound data changed");
   }

   private void OnCommit(Object src, EventArgs ea){
      FinalizeEdit();
      adapter.Update(emps);
   }

   private void ReadEmployees
    (string pathToAccessDB){
      OleDbConnection cnctn = new OleDbConnection();
      cnctn.ConnectionString=
      "Provider=Microsoft.JET.OLEDB.4.0;" +
      "data source=" + pathToAccessDB;
      cnctn.Open();

      string selStr = "SELECT * FROM EMPLOYEES";
      adapter = new OleDbDataAdapter(selStr, cnctn);
      new OleDbCommandBuilder(adapter);

      emps = new DataSet("Employees");
      adapter.Fill(emps);
   }

   public static void Main(){
```

```
        Application.Run(new FormEdit());
    }
}//:~
```

The **FormEdit** class has several instance variables: **colForLabel** is a **Hashtable** that is used to establish a correspondence between a **DataSet** column name and a **Label** control. The next 3 instance variables, **editWindowActive**, **editWindow**, and **labelBeingEdited** are used in the "dynamic" editing scheme – all editing is done in the **editWindow TextBox**, while a reference to the editing target is held in the **labelBeingEdited** variable. The **adapter** and **emps** variables are the familiar variables holding the **IDataAdapter** and **DataSet** associated with the Northwind database.

The **FormEdit( )** constructor initializes **emps** and **adapter** in **ReadEmployees( )**, which does not differ from the previous examples. **InitDataStructure( )** initializes the **colForLabel** data structure; an anonymous **Label** is used as the key and the column's name is the value. One can imagine a more sophisticated version of this method that iterates over the **DataSet**, creating as many **Control**s as are necessary.

InitLabels( ) is responsible for setting up the display of the data structure initialized in **InitDataStructure( )**. Because the .NET Framework does not have a bidirectional version of the **IDictionary** interface that can look up a key based on the value, we have to use a nested loop to find the **Label** associated with the **colName**. Once found, **InitLabels( )** calls **InitLabel( )** to configure the specific **Label**. Again, a more sophisticated version of this method might do a more sophisticated job of laying out the display.

InitLabel( ) is responsible for wiring up the **Label** to the **DataSet**'s column name. Two event handlers are added to the **Label**: **OnLabelClick( )** and **OnTextChange( )**. The method also associates **OnBoundDataChange( )** with the **Binding**'s **Parse** event, which occurs when the value of a databound control changes. The **Binding** is created from the column name that was passed in as an argument and associated with the **Label** that was also passed in.

The last method called by the constructor is **InitCommitter( ),** which initializes a **Button** that will be used to trigger a database update.

After the constructor runs, the **DataSet emps** is filled with Northwind's employee data and the **Label**s on the **EditForm** show the first and last name of the first record (this example doesn't have any navigation). When the user clicks on one of these labels, they activate **OnLabelClick( )**.

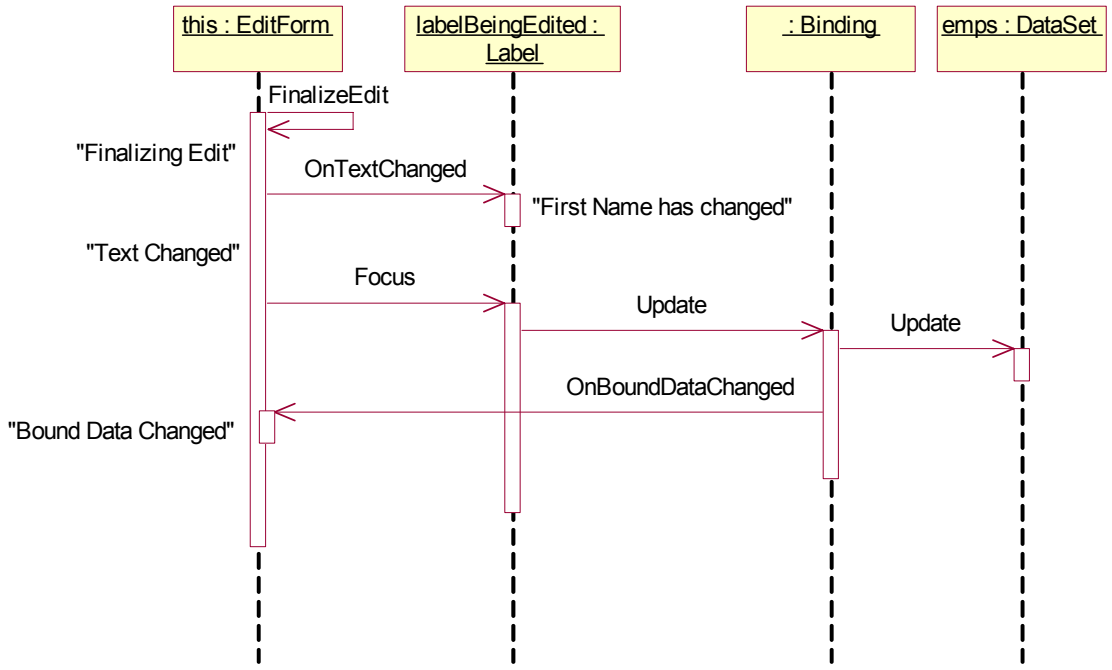
All **Label**s on the form share the **OnLabelClick( )** event handler, but simply casting the **src** to **Label** gives us the needed information to manipulate the **editWindow**. The first time through **OnLabelClick( )**, the **editWindowActive** Boolean will be **false**, so let's momentarily delay discussion of the **FinalizeEdit( )** method.

Our dynamic editing technique is to place the editing control on the UI at the appropriate place, and then to associate the editing control with the underlying data. **PlaceEditWindowOverLabel( )** sets the **editWindow** to overlay the **Label** the user clicked on, makes it visible, and requests the focus. **AssociateEditorWithLabel( )** sets the text of the **editorWindow** to what is in the underlying **Label** and sets the **labelBeingEdited** instance variable to the clicked-on **Label**. The visual effect of all this is that when someone clicks on either **Label**, it appears as if the **Label** turns into an **EditBox**. Again, this is no big deal with two labels on the form, but if there were a couple dozen fields beings displayed on the screen, there can be a significant resource and speed improvement by having a single, dynamic editing control.

Since **PlaceEditWindowOverLabel( )** set **editWindowActive** to **true**, subsequent calls to **OnLabelClick( )** will call **FinalizeEdit( )**. **FinalizeEdit( )** puts the text from the **editWindow** into the underlying **labelBeingEdited**. This alone *does not update* the **DataBinding**, you must give the label the focus, even for a moment, in order to update the data. **OnBoundDataChanged( )** is called subsequent to the data update, and when run, you'll see:

```
Finalizing edit
FirstName has changed
Text changed
Bound data changed
```

Indicating this sequence:



The above diagram is a UML *activity diagram* and is one of the most helpful diagrams for whiteboard sessions (as argued effectively by Scott Ambler at www.agilemodiling.org, you will derive more benefit from low-tech modeling with others than you can ever derive from high-tech modeling by yourself).

The activity diagram has time on the vertical axis and messages and objects on the horizontal. The boxes on the vertical lines correspond to method invocations and the call stack. Thus, **this.OnBoundDataChanged( )** is called from an anonymous **BindingObject**'s **Update( )**[2] method, which is called from **labelBeingEdited.Focus(),** which we call in **this.FinalizeEdit( )**.

Although **FinalizeEdit( )** modifies the **DataSet emps**, the call to **adapter.Update( )** in **OnCommit( )** is required to actually move the

[2] Actually, the names and sequence of events that happen in **Control.Focus( )** is considerably more complex than what is portrayed here, but glossing over details not relevant to the task at hand is one of the great benefits of diagramming. This is one of the reasons diagramming tools that dynamically bind to actual source code are often less useful than, say, a piece of paper.

data back into the database. Be sure to be familiar with ADO.NET's optimistic concurrency behavior, as discussed in chapter #ado.net# before engaging in any database projects.

Menus

User interface experts say that menus are overrated. Programmers, who are trained to think in terms of hierarchies, and whose careers are computer-centric, don't bat an eye at cascading menu options and fondly recall the DOS days when you could specify incredibly complex spreadsheet and word processing behaviors by rattling off the first letters of menu sequences (nowadays, you have to say "So are you seeing a dialog that says 'Display Properties' and that has a bunch of tabs? Okay, find the tab that says 'Appearance' and click on it. Do you see the button that reads 'Advanced'? It should be in the lower-right corner…"). For many users, though, menus are surprisingly difficult to navigate. Nevertheless, unless you're creating a purely Web-based interface, the odds are quite good that you'll want to add menus and context menus to your UIs.

There are two types of menu in Windows Forms, a **MainMenu** that displays across the top of a **Form**, and a **ContextMenu** that can be associated with any **Control** and is typically displayed on a right-button mouse click over the **Control**.

Both forms of menu contain **MenuItem**s and **MenuItem**s may, in turn, contain other **MenuItem**s (once again, the familiar containment model of Windows Forms) . When selected, the **MenuItem** triggers a **Click** event, even if the **MenuItem** was chosen via a shortcut or access key. This example shows both types of menus and how to create cascading menus:

```
//:c13:MenuDemo.cs
//Demonstrates menus

using System;
using System.Drawing;
using System.Windows.Forms;

class MenuDemo : Form {
```

```
MenuDemo(){
    Text = "Menu Demo";
    MainMenu courseMenu = new MainMenu();

    MenuItem appetizers = new MenuItem();
    appetizers.Text = "&Appetizers";
    MenuItem[] starters = new MenuItem[3];
    starters[0] = new MenuItem();
    starters[0].Text = "&Pot stickers";
    starters[1] = new MenuItem();
    starters[1].Text = "&Spring rolls";
    starters[2] = new MenuItem();
    //Note escaped "&"
    starters[2].Text = "&Hot && Sour Soup";
    appetizers.MenuItems.AddRange(starters);
    foreach(MenuItem i in starters){
      i.Click +=
       new EventHandler(OnCombinableMenuSelected);
    }

    MenuItem mainCourse = new MenuItem();

    mainCourse.Text = "&Main Course";
    MenuItem[] main = new MenuItem[4];
    main[0] = new MenuItem();
    main[0].Text = "&Sweet && Sour Pork";
    main[1] = new MenuItem();
    main[1].Text = "&Moo shu";
    main[2] = new MenuItem();
    main[2].Text = "&Kung Pao Chicken";
    //Out of Kung Pao Chicken
    main[2].Enabled = false;
    main[3] = new MenuItem();
    main[3].Text = "General's Chicken";
    mainCourse.MenuItems.AddRange(main);
    foreach(MenuItem i in main){
      i.RadioCheck = true;
      i.Click +=
        new EventHandler(OnExclusiveMenuSelected);
    }
```

```
        MenuItem veg = new MenuItem();
        veg.Text = "Vegetarian";
        veg.RadioCheck = true;
        veg.Click +=
           new EventHandler(OnExclusiveMenuSelected);
        MenuItem pork = new MenuItem();
        pork.Text = "Pork";
        pork.RadioCheck = true;
        pork.Click +=
           new EventHandler(OnExclusiveMenuSelected);
        main[1].MenuItems.AddRange(
           new MenuItem[]{veg,pork});

        courseMenu.MenuItems.Add(appetizers);
        courseMenu.MenuItems.Add(mainCourse);

        ContextMenu contextMenu = new ContextMenu();
        foreach(MenuItem a in starters){
            contextMenu.MenuItems.Add(a.CloneMenu());
        }
        contextMenu.MenuItems.Add(
           new MenuItem().Text = "-");
        foreach(MenuItem m in main){
            contextMenu.MenuItems.Add(m.CloneMenu());
        }
        Menu = courseMenu;
        ContextMenu = contextMenu;
    }

    private void OnCombinableMenuSelected
       (object sender, EventArgs args){
        MenuItem selection = (MenuItem) sender;
        selection.Checked = !selection.Checked;
    }

    private void OnExclusiveMenuSelected
       (object sender, EventArgs args){
        MenuItem selection = (MenuItem) sender;
        bool selectAfterClear = !selection.Checked;
        //Must implement radio-button functionality
programmatically
```

```
        Menu parent = selection.Parent;
        foreach(MenuItem i in parent.MenuItems){
            i.Checked = false;
        }
        selection.Checked = selectAfterClear;
    }


    public static void Main(){
        Application.Run(new MenuDemo());
    }
}//:~
```

The **MenuDemo( )** constructor creates a series **MenuItem**s. The ampersand (&) in the **MenuItem.Text** property sets the accelerator key for the item (so "Alt-A" will activate the "Appetizers" menu and "Alt-M" the "Main Course" menu). To include an ampersand in the menu text, you must use a double ampersand (for instance, "**Hot && Sour Soup**").

Each **MenuItem** created is added to either the **mainMenu**.**MenuItems** collection or to the **MenuItems** collection of one of the other **MenuItem**s.

The **MenuItem**s in the **appetizers Menu** have the default **false** value for their **RadioCheck** property. This means that if their **Selection.Checked** property is set to **true**, they will display a small checkmark. The **MenuItem**s in **mainCourse** have **RadioCheck** set to **true**, and when they have **Selected.Checked** set to true, they display a small circle. However, this is a display option only, the mutual exclusion logic of a radio button must be implemented by the programmer, as is shown in the **OnExclusiveMenuSelected( )** method.

A **MenuItem** is deactivated by setting its **Enabled** property to false, as is done with the "Kung Pao Chicken" entrée.

To duplicate a **Menu,** you use not **Clone( )** but **CloneMenu( )**, as shown in the loops that populate the **contextMenu**. The **contextMenu** also demonstrates that a **MenuItem** with its **Text** property set to a single dash is displayed as a separator bar in the resulting menu.

Standard Dialogs

Windows Forms provides several standard dialogs both to ease common chores and maintain consistency for the end-user. These dialogs include file open and save, color choice, font dialogs, and print preview and print dialogs. We're going to hold off on the discussion of the printing dialogs until the section on printing in the GDI+ chapter, but this example shows the ease with which the others are used:

```
//:c13:StdDialogs.cs
using System;
using System.Drawing;
using System.Windows.Forms;

class StdDialogs : Form{
   Label label = new Label();

   StdDialogs(){
      MainMenu menu = new MainMenu();
      Menu = menu;

      MenuItem fMenu = new MenuItem("&File");
      menu.MenuItems.Add(fMenu);

      MenuItem oMenu = new MenuItem("&Open...");
      oMenu.Click += new EventHandler(OnOpen);
      fMenu.MenuItems.Add(oMenu);

      MenuItem cMenu = new MenuItem("&Save...");
      cMenu.Click += new EventHandler(OnClose);
      fMenu.MenuItems.Add(cMenu);
      fMenu.MenuItems.Add(new MenuItem("-"));

      MenuItem opMenu = new MenuItem("&Options");
      menu.MenuItems.Add(opMenu);

      MenuItem clrMenu = new MenuItem("&Color...");
      clrMenu.Click += new EventHandler(OnColor);
      opMenu.MenuItems.Add(clrMenu);
```

```
      MenuItem fntMenu = new MenuItem("&Font...");
      fntMenu.Click += new EventHandler(OnFont);
      opMenu.MenuItems.Add(fntMenu);

      label.Text = "Some text";
      label.Dock = DockStyle.Fill;
      Controls.Add(label);
   }

   public void OnOpen(object src, EventArgs ea){
      OpenFileDialog ofd = new OpenFileDialog();
      ofd.Filter =
         "C# files (*.cs)|*.cs|All files (*.*)|*.*";
      ofd.FilterIndex = 2;

      DialogResult fileChosen = ofd.ShowDialog();
      if(fileChosen == DialogResult.OK){
         foreach(string fName in ofd.FileNames){
            Console.WriteLine(fName);
         }
      }else{
         System.Console.WriteLine("No file chosen");
      }
   }

   public void OnClose(object src, EventArgs ea){
      SaveFileDialog sfd = new SaveFileDialog();
      DialogResult saveChosen = sfd.ShowDialog();
      if(saveChosen == DialogResult.OK){
          Console.WriteLine(sfd.FileName);
      }
   }

   public void OnColor(object src, EventArgs ea){
       ColorDialog cd = new ColorDialog();
       if(cd.ShowDialog() == DialogResult.OK){
           Color c = cd.Color;
           label.ForeColor = c;
           Update();
       }
   }
```

```
    public void OnFont(object src, EventArgs ea){
        FontDialog fd = new FontDialog();
        if(fd.ShowDialog() == DialogResult.OK){
            Font f = fd.Font;
            label.Font = f;
            Update();
        }
    }

    public static void Main(){
       Application.Run(new StdDialogs());
    }
}//:~
```

The **StdDialogs( )** constructor initializes a menu structure and places a **Label** on the form. **OnOpen( )** creates a new **OpenFileDialog** and sets its **Filter** property to show either all files or just those with **.cs** extensions. File dialog filters are confusing. They are specified with a long string, delimited with the pipe symbol (|). The displayed names of the filters are placed in the odd positions, while the filters themselves are placed in the even positions. The **FilterIndex** is *one-based!* So by setting it's value to 2, we're settings its value to the *.* filter, the fourth item in the list.

Like all the dialogs, the **OpenFileDialog** is shown with the **ShowDialog( )** method. This opens the dialog in *modal* form; the user must close the modal dialog before returning to any other kind of input to the system. When the dialog is closed, it returns a value from the **DialogResult** enumeration. **DialogResult.OK** is the hoped-for value; others are:

- Abort
- Cancel
- Ignore
- No
- None

- Retry
- Yes

Obviously, not all dialogs will return all (or most) of these values.

Both the **OpenFileDialog** and the **SaveFileDialog** have **OpenFile( )** methods that open the specified file (for reading or writing, respectively), but in the example, we just use the **FileNames** and **FileName** properties to get the list of selected files (in **OnOpen( )**) and the specified file to be written to in **OnSave( )**. The **SaveFileDialog.OverwritePrompt** property, which defaults to **true**, specifies whether the dialog will automatically ask the user "Are you sure...?"

OnColor( ) and **OnFont( )** both have appropriate properties (**Color** and **Font**) that can be read after the dialogs close. The **label** is changed to use that value and then the **Update( )** method is called in order to redraw and relayout the **StdDialogs** form.

Usage-Centered Design

In the discussion of GUI Architectures, we mentioned the adage that "To the end user, the UI *is* the application." This is one of the tenets of *usage-centered design*, an approach to system development that fits very well with the move towards *agile development* that propose that shorter product cycles (as short as a few weeks for internal projects) that are intensely focused on delivering requested end-user value are much more successful than the traditional approach of creating a balance of functional, non-functional, and market-driven features that are released in "major roll-outs."

Far too many discussions of software development fail to include the end-user, much less accord them the appropriate respect as the driver of what the application should do and how it should appear. Imagine that one day you were given a survey on your eating habits as part of a major initiative to provide you a world-class diet. Eighteen months later, when you'd forgotten about the whole thing, you're given a specific breakfast, lunch, and dinner and told that this was all you could eat for the next eighteen

months. And that the meals were prepared by dieticians, not chefs, and that the meals had never actually been tasted by anyone. That's how most software is developed.

No interface library is enough to make a usable interface. You *must* actively work with end users to discover their needs (you'll often help *them* discover the words to express a need they assumed could not be fixed) , you *must* silently watch end-users using your software (a humbling experience), and you *must* evolve the user interface based on the needs of the user, not the aesthetic whims of a graphic designer[3].

One of us (Larry) had the opportunity to work on a project that used usage-centered design to create new types of classroom management tools for K-12 teachers (a group that is rightfully distrustful of the technical "golden bullets" that are regularly foisted upon them). The UI had lots of standard components, and three custom controls. Two of the custom controls were *never noticed* by end-users (they just used them, not realizing that they were seeing complex, non-standard behavior). We could always tell when users saw the third, though, because they literally *gasped* when they saw it. It was a heck of a good interface[4].

Summary

C# has an object-oriented bound method type called *delegate*. Delegates are first-class types, and can be instantiated by any method whose signature exactly matches the delegate type.

There are several architectures that may be used to structure the GUI, either as an independent subsystem or for the program as a whole. The Visual Designer tool in Visual Studio .NET facilitates an architecture

---

[3] This is not to disparage graphic designers or their work. But creating a usable interface is *nothing* like creating a readable page or an eye-catching advertisement. If you can't get a designer with a background in Computer-Human Interaction (CHI), at least use a designer with a background in industrial design.

[4] Unfortunately, the CEO saw fit to spend our $15,000,000 in venture capital on prostitutes, drugs, and a Jaguar sedan with satellite navigation for the company car. Oh yeah, we also had free soda.

called Form-Event-Control, which provides a minimal separation of the interface from the domain logic. Model-View-Controller provides the maximum separation of interface, input, and domain logic, but is often more trouble than its worth. Presentation-Abstraction-Control is an architecture that creates self-contained components that are responsible for both their own display and domain logic; it is often the best choice for working in Windows Forms.

Windows Forms is a well-architected class library that simplifies the creation of the vast majority of user interfaces. It is programmed by a series of public delegate properties called *events* that are associated with individual **Control**s. **Control**s contain other **Control**s. A **Form** is a type of **Control** that takes the form of a window.

Controls are laid out within their containing **Control** by means of the **Location**, **Dock**, and **Anchor** properties. This layout model is simple, but not simplistic, and if combined with a proper attention to GUI architecture, can lead to easily modified, easy-to-use user interfaces.

Properties of **Control**s can be bound to values within a **DataSet** via a collection of **Binding**s. Often, the bound property is the "Text" property of the **Control**. A collection of **Binding**s may be coordinated by a **BindingManagerBase** provided by a **BindingContext**. Such coordinated **Binding**s will refer to a single data record or set of properties and thus, **Control**s bound to those **Binding**s will simultaneously update.

While Windows Forms is well-architected, there are many quirks and inconsistencies in the various **Control**s. These quirks range from what are clearly defects (**CheckedListBox.CheckedIndexCollection** include items whose checkstate is **indeterminate**), to design flaws (ther should be a **Control.Tooltip** property), to undocumented behaviors (when handed an object of illegal type, should **IDataObject.SetObject( )** throw an exception?), to behaviors that reflect an underlying quirk in the operating system (the string that specifies **OpenFileDialog.Filter**).

Visual Studio .NET makes the creation of Windows Forms interface very easy but is no substitute for working directly with the implementing code. Most real applications will use a combination of Visual Designer-

generated code and hand-written implementations. One way or the other, the success of your application is dependent on the usability of your interface; if at all possible, work with a computer-human interface specialist to guide the creation of your UI.

Exercises

14: GDI+ Overview

While Windows Forms provides a great basis for some large majority of user interfaces, the .NET Framework exposes

Drawing pixels
Drawing shapes
Filling and stroking
Printing

Demonstrates common dialogs skipped in previous discussion

```
//:c13:Printing.cs
//Demonstrates printing from Windows Forms
using System;
using System.Drawing;
using System.Drawing.Printing;
using System.Windows.Forms;

class Printing : Form {
   PaperWaster pw = new PaperWaster();
   Printing(){
      MainMenu menu = new MainMenu();
      Menu = menu;

      MenuItem fMenu = new MenuItem("&File");
      menu.MenuItems.Add(fMenu);
      MenuItem prvMenu = new MenuItem("P&review...");
      fMenu.MenuItems.Add(prvMenu);
      prvMenu.Click += new EventHandler(OnPreview);
```

```
      MenuItem prtMenu = new MenuItem("&Print...");
      fMenu.MenuItems.Add(prtMenu);
      prtMenu.Click += new EventHandler(OnPrint);
   }

   public void OnPreview(object src, EventArgs ea){
      PrintPreviewDialog ppd =
         new PrintPreviewDialog();
      ppd.Document = pw.Document;
      if(ppd.ShowDialog() == DialogResult.OK){

      }
   }

   public void OnPrint(object src, EventArgs ea){
      PrintDialog pd = new PrintDialog();
      pd.Document = pw.Document;
      if(pd.ShowDialog() == DialogResult.OK){
      }
   }

   public static void Main(){
      Application.Run(new Printing());
   }
}

class PaperWaster{
   PrintDocument pd = new PrintDocument();
   internal PrintDocument Document{
      get { return pd; }
   }

   internal PaperWaster(){
      pd.PrintPage +=
         new PrintPageEventHandler(PrintAPage);
   }

   void PrintAPage(object src, PrintPageEventArgs ea){
      Graphics g = ea.Graphics;
      Font f = new Font("Arial", 36);
      SolidBrush b = new SolidBrush(Color.Black);
```

```
        PointF p = new PointF(10.0f, 10.0f);
        g.DrawString("Reduce, Reuse, Recycle", f, b, p);
    }

}//:~
```

Accessing DirectX

Creating a screensvaer

Creating a system service

Usage-Centered Design

Creating an application (Windows & Menus)

- Ambient Properties
- Application
- ApplicationContext
- AxHost
- ErrorProvider
- FeatureSupport
- Help
- Message
- MessageBox
- NotifyIcon

Progress Bar
PropertyGrid
SelectionRange
TabControl/ Tabbed Pages
Timer
ToolBar
TrackBar
UserControl
Windows Services
Programming techniques
Binding events dynamically
Separating business logic from UI logic
Visual programming
Summary
Exercises

14: Multithreaded Programming

Objects divide the solution space into logical chunks of state and behavior. Often, you need groups of your objects to perform their behavior simultaneously, as independent subtasks that make up the whole.

Each of these independent subtasks is called a *thread*, and you program as if each thread runs by itself and has the CPU to itself. Some underlying mechanism is actually dividing up the CPU time for you, but in general, you don't have to think about it, which makes programming with multiple threads a much easier task.

A *process* is a self-contained running program with its own address space. A *multitasking* operating system is capable of running more than one process (program) at a time, while making it look like each one is chugging along on its own, by periodically providing CPU cycles to each process. A thread is a single sequential flow of control within a process. A single process can thus have multiple concurrently executing threads.

There are many possible uses for multithreading, but in general, you'll have some part of your program tied to a particular event or resource, and you don't want to hang up the rest of your program because of that. So you create a thread associated with that event or resource and let it run independently of the main program. A good example is a "cancel" button to stop a lengthy calculation —you don't want to be forced to poll the cancel button in every piece of code you write in your program and yet you want the cancel button to be responsive, as if you *were* checking it regularly. In fact, one of the most immediately compelling reasons for multithreading is to produce a responsive user interface.

.NET’s Threading Model
Thread Scheduling
Threading Problems
The Cardinal Rules of Threading
Thread Lifecycle
Starting Threads
Stopping Threads
Pausing and Restarting
Blocking and Waiting
Exception Handling in Threads
Threads and Interoperability

Threads and Garbage Collection

Threads and Scalability

Responsive user interfaces

As a starting point, consider a program that performs some CPU-intensive operation and thus ends up ignoring user input and being unresponsive. This one simply covers the display in a random patchwork of red spots:

```
//<example>
///<chapter>Multithreaded</chapter>
///<program>Counter1.cs</program>
// A non-responsive user interface.
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Threading;

class Counter1 : Form {
    int numberToCountTo;

    Counter1(int numberToCountTo) {
        this.numberToCountTo = numberToCountTo;
        ClientSize = new System.Drawing.Size(500, 300);
        Text = "Nonresponsive interface";

        Button start = new Button();
        start.Text = "Start";
        start.Location = new Point(10, 10);
        start.Click += new EventHandler(StartCounting);

        Button onOff = new Button();
        onOff.Text = "Toggle";
        onOff.Location = new Point(10, 40);
        onOff.Click += new EventHandler(StopCounting);
```

```
        Controls.AddRange(new Control[]{ start, onOff });
    }

    public void StartCounting(Object sender, EventArgs
args) {
        Rectangle bounds = Screen.GetBounds(this);
        int width = bounds.Width;
        int height = bounds.Height;

        Graphics g = Graphics.FromHwnd(this.Handle);
        Pen pen = new Pen(Color.Red, 1);

        Random rand = new Random();
        runFlag = true;
        for (int i = 0; runFlag && i < numberToCountTo;
i++) {
            //Do something mildly time-consuming
            int x = rand.Next(width);
            int y = rand.Next(height);
            g.DrawRectangle(pen, x, y, 1, 1);
            Thread.Sleep(10);
        }
    }

    bool runFlag = true;

    public void StopCounting(Object sender, EventArgs
args){
        runFlag = false;
    }

    public static void Main() {
        Application.Run(new Counter1(10000));
    }
} //</example>
```

At this point, the graphics code should be reasonably familiar from Chapter 13. The **startCounting( )** method is where the program stays busy: it loops **numberToCountTo** times, picks a random spot on the form, and draws a 1-pixel rectangle their.

Part of the loop inside **startCounting( )** calls **Thread.Sleep( )**. . **Thread.Sleep()** pauses the currently executing thread for some amount of milliseconds. Regardless of whether you're explicitly using threads, you can produce the current thread used by your program with **Thread** and the static **Sleep( )** method.

When the **Start** button is pressed, **startCounting( )** is invoked. On examining **startCounting( )**, you might think that it should allow multithreading because it goes to sleep. That is, while the method is asleep, it seems like the CPU could be busy monitoring other button presses. But it turns out that the real problem is that **startCounting( )** doesn't return until *after* its finished, and this means that **stopCounting( )** is never called until it's too late for its behavior to be meaningful. Since you're stuck inside **startCounting( )** for the first button press, the program can't handle any other events. (To get out, you must either wait until **startCounting()** ends, or kill the process; the easiest way to do this is to press Control-C or to click a couple times to trigger Windows "Program Not Responding" dialogue.)

The basic problem here is that **startCounting( )** needs to continue performing its operations, and at the same time it needs to return so that **stopCounting( )** can be activated and the user interface can continue responding to the user. But in a conventional method like **startCounting( )** it cannot continue *and* at the same time return control to the rest of the program. This sounds like an impossible thing to accomplish, as if the CPU must be in two places at once, but this is precisely the illusion that threading provides.

The thread model (and its programming support in C#) is a programming convenience to simplify juggling several operations at the same time within a single program. With threads, the CPU will pop around and give each thread some of its time. Each thread has the consciousness of constantly having the CPU to itself, but the CPU's time is actually sliced between all the threads. The exception to this is if your program is running on multiple CPUs. But one of the great things about threading is that you are abstracted away from this layer, so your code does not need to know whether it is actually running on a single CPU or many. Thus, threads are a way to create transparently scalable programs.

Threading reduces computing efficiency somewhat, but the net improvement in program design, resource balancing, and user convenience is often quite valuable. Of course, if you have more than one CPU, then the operating system can dedicate each CPU to a set of threads or even a single thread and the whole program can run much faster. Multitasking and multithreading tend to be the most reasonable ways to utilize multiprocessor systems.

Creating Threads

To use Threads in C#, you simply create a **Thread** object and a delegate called **ThreadStart**. **Thread** takes cares of the underlying creation and management of the thread, while **ThreadStart** specifies the actual code that will be executed when the thread is active. Thus, whatever method is passed to the **ThreadStart** constructor is the code that will be executed "simultaneously" with the other threads in a program.

The following example creates any number of threads that it keeps track of by assigning each thread a unique number, generated with a **static** variable. The delegated behavior is in the **run( )** method, which is overridden to count down each time it passes through its loop and finishes when the count is zero (at the point when the delegated method returns, the thread is terminated).

```
//<example>
///<chapter>c14</chapter>
///<program>SimpleThreading.cs</program>
// Very simple Threading example.
using System.Threading;

public class SimpleThreading  {
  private int countDown = 5;
  private static int threadCount = 0;
  private int threadNumber = ++threadCount;
  public SimpleThreading() {
    System.Console.WriteLine("Making " + threadNumber);
  }
  public void run() {
    while(true) {
      System.Console.WriteLine("Thread " +
```

```
        threadNumber + "(" + countDown + ")");
      if(--countDown == 0) return;
    }
  }
  public static void Main() {
    for(int i = 0; i < 5; i++){
        SimpleThreading st = new SimpleThreading();
        Thread aThread = new Thread( new
ThreadStart(st.run));
        aThread.Start();
    }
    System.Console.WriteLine("All Threads Started");
  }
} //</example>
```

A thread-delegate method (often called **run( )**) virtually always has some kind of loop that continues until the thread is no longer necessary, so you must establish the condition on which to break out of this loop (or, in the case above, simply **return** from **run( )**). Often, **run( )** is cast in the form of an infinite loop, which means that, barring some external factor that causes **run( )** to terminate, it will continue forever.

In **Main( )** you can see a number of threads being created and run. The **Start( )** method in the **Thread** class performs the initialization for the thread and then calls **run( )**. So the steps are: the constructor is called to build the object that will do the work, the **ThreadStart** delegate is given the name of the working function, the **ThreadStart** is passed to a newly created **Thread**, then **Start( )** configures the thread and calls the delegated function -- **run( )**. If you don't call **Start( )**, the thread will never be started.

The output for one run of this program (it will be different from one run to another) is:

```
Making 1
Making 2
Making 3
Thread 1(5)
Thread 1(4)
Thread 1(3)
Thread 1(2)
Thread 1(1)
```

```
Making 4
Making 5
All Threads Started
Thread 2(5)
Thread 2(4)
Thread 2(3)
Thread 2(2)
Thread 2(1)
Thread 5(5)
Thread 5(4)
Thread 5(3)
Thread 5(2)
Thread 5(1)
Thread 4(5)
Thread 4(4)
Thread 4(3)
Thread 4(2)
Thread 4(1)
Thread 3(5)
Thread 3(4)
Thread 3(3)
Thread 3(2)
Thread 3(1)
```

You'll notice that nowhere in this example is **Thread.Sleep( )** called, and yet the output indicates that each thread gets a portion of the CPU's time in which to execute. This shows that **Sleep( )**, while it relies on the existence of a thread in order to execute, is not involved with either enabling or disabling threading. It's simply another method.

You can also see that the threads are not run in the order that they're created. In fact, the order that the CPU attends to an existing set of threads is indeterminate, unless you go in and adjust the Priority property of the thread.

When **Main( )** creates the **Thread** objects it isn't capturing the references for any of them. @todo: Check this-> An ordinary object would be fair game for garbage collection, but not a **Thread**. Each **Thread** "registers" itself so there is actually a reference to it someplace and the garbage collector can't clean it up.

Threading for a responsive interface

Now it's possible to solve the problem in **Counter1.cs** with a thread. The trick is to make the working method —that is, the loop that's inside **stopCounting( )**—a delegate of a thread. When the user presses the **start** button, the thread is started, but then the *creation* of the thread completes, so even though the thread is running, the main job of the program (watching for and responding to user-interface events) can continue. Here's the solution:

```
//<example>
///<chapter>Multithreaded</chapter>
///<program>Counter2.cs</program>
// A non-responsive user interface.
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Threading;

class Counter2 : Form {
    int numberToCountTo;

    Counter2(int numberToCountTo) {
        this.numberToCountTo = numberToCountTo;
        ClientSize = new System.Drawing.Size(500, 300);
        Text = "Responsive interface";

        Button start = new Button();
        start.Text = "Start";
        start.Location = new Point(10, 10);
        start.Click += new EventHandler(StartCounting);

        Button onOff = new Button();
        onOff.Text = "Toggle";
        onOff.Location = new Point(10, 40);
        onOff.Click += new EventHandler(StopCounting);

        Controls.AddRange(new Control[]{ start, onOff });
    }
```

```
    public void StartCounting(Object sender, EventArgs
args) {
        ThreadStart del = new ThreadStart(paintScreen);
        Thread t = new Thread(del);
        t.Start();
    }

    public void paintScreen(){
        Rectangle bounds = Screen.GetBounds(this);
        int width = bounds.Width;
        int height = bounds.Height;

        Graphics g = Graphics.FromHwnd(this.Handle);
        Pen pen = new Pen(Color.Red, 1);

        Random rand = new Random();
        runFlag = true;
        for (int i = 0; runFlag && i < numberToCountTo;
i++) {
            //Do something mildly time-consuming
            int x = rand.Next(width);
            int y = rand.Next(height);
            g.DrawRectangle(pen, x, y, 1, 1);
            Thread.Sleep(10);
        }
    }

    bool runFlag = true;

    public void StopCounting(Object sender, EventArgs
args){
        runFlag = false;
    }

    public static void Main() {
        Application.Run(new Counter2(10000));
    }
} //</example>
```

Counter2 is a straightforward program, whose only job is to set up and maintain the user interface. But now, when the user presses the **start** button, the event-handling code does not do the time-consuming work. Instead a thread is created and started, and then the **Counter2** interface can continue to respond to events.

When you press the **onOff** button it toggles the **runFlag** inside the **Counter2** object. Then, when the "worker" thread calls **paintScreen()**, it can look at that flag and decide whether to continue or stop. Pressing the **onOff** button produces an apparently instant response. Of course, the response isn't really instant, not like that of a system that's driven by interrupts. The painting stops only when the thread has the CPU and notices that the flag has changed.

Sharing limited resources

You can think of a single-threaded program as one lonely entity moving around through your problem space and doing one thing at a time. Because there's only one entity, you never have to think about the problem of two entities trying to use the same resource at the same time, like two people trying to park in the same space, walk through a door at the same time, or even talk at the same time.

With multithreading, things aren't lonely anymore, but you now have the possibility of two or more threads trying to use the same limited resource at once. Colliding over a resource must be prevented or else you'll have two threads trying to access the same bank account at the same time, print to the same printer, or adjust the same valve, etc.

Improperly accessing resources

Consider a variation on the counters that have been used so far in this chapter. In the following example, each thread contains two counters that are incremented and displayed inside **run( )**. In addition, there's another thread of class **Watcher** that is watching the counters to see if they're always equivalent. This seems like a needless activity, since looking at the code it appears obvious that the counters will always be the same. But that's where the surprise comes in. Here's the first version of the program:

```
//<example>
///<chapter>c14</chapter>
///<program>Sharing1.cs</program>
// Problems with resource sharing while threading.
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Threading;

public class Sharing1 : Form {
    private TextBox accessCountBox = new TextBox();
    private Button start = new Button();
    private Button watch = new Button();

    private int accessCount = 0;
    public void incrementAccess() {
        accessCount++;
        accessCountBox.Text = accessCount.ToString();
    }

    private int numCounters = 12;
    private int numWatchers = 15;

    private TwoCounter[] s;

    public Sharing1() {
        ClientSize = new Size(450, 480);
        Panel p = new Panel();
        p.Size = new Size(400, 50);

        start.Click += new EventHandler(StartAllThreads);
        watch.Click += new
EventHandler(StartAllWatchers);

        accessCountBox.Text = "0";
        accessCountBox.Location = new Point(10, 10);
        start.Text = "Start threads";
        start.Location = new Point(110, 10);
        watch.Text = "Begin watching";
        watch.Location = new Point(210, 10);
```

```
        p.Controls.Add(start);
        p.Controls.Add(watch);
        p.Controls.Add(accessCountBox);

        s = new TwoCounter[numCounters];
        for (int i = 0; i < s.Length; i++) {
            s[i] = new TwoCounter(new
TwoCounter.IncrementAccess(incrementAccess));
            s[i].Location = new Point(10, 50 +
s[i].Height * i);
            Controls.Add(s[i]);
        }

        this.Closed += new EventHandler(StopAllThreads);

        Controls.Add(p);
    }

    public void StartAllThreads(Object sender, EventArgs
args) {
        for (int i = 0; i < s.Length; i++)
            s[i].Start();
    }

    public void StopAllThreads(Object sender, EventArgs
args){
        for(int i = 0; i < s.Length; i++){
            if(s[i] != null){
                s[i].Stop();
            }
        }
    }

    public void StartAllWatchers(Object sender, EventArgs
args) {
        for (int i = 0; i < numWatchers; i++)
            new Watcher(s);
    }

    public static void Main(string[] args) {
        Sharing1 app = new Sharing1();
```

```
        if (args.Length > 0) {
            app.numCounters = SByte.Parse(args[0]);
            if (args.Length == 2) {
                app.numWatchers = SByte.Parse(args[1]);
            }
        }
        Application.Run(app);
    }
}

    class TwoCounter : Panel {
        private bool started = false;
        private Label t1;
        private Label t2;
        private Label lbl;
        private Thread t;

        private int count1 = 0, count2 = 0;
        public delegate void IncrementAccess();
        IncrementAccess del;

        // Add the display components
        public TwoCounter(IncrementAccess del) {
            this.del = del;

            this.Size = new Size(350, 30);
            this.BorderStyle = BorderStyle.Fixed3D;
            t1 = new Label();
            t1.Location = new Point(10, 10);
            t2 = new Label();
            t2.Location = new Point(110, 10);
            lbl = new Label();
            lbl.Text = "Count1 == Count2";
            lbl.Location = new Point(210, 10);
            Controls.AddRange(new Control[]{t1, t2,
lbl});

            //Initialize the Thread
            t = new Thread(new ThreadStart(run));
        }
        public void Start() {
```

```
            if (!started) {
                started = true;
                t.Start();
            }
        }

        public void Stop(){
            t.Abort();
        }
        public void run() {
            while (true) {
                t1.Text = ++count1.ToString();
                t2.Text = ++count2.ToString();
                Thread.Sleep(500);
            }
        }
        public void synchTest() {
            del();
            if (count1 != count2)
                lbl.Text = "Unsynched";
        }
    }

    class Watcher {
        TwoCounter[] s;

        public Watcher(TwoCounter[] s) {
            this.s = s;
            new Thread(new ThreadStart(run)).Start();
        }
        public void run() {
            while (true) {
                for (int i = 0; i < s.Length; i++)
                    s[i].synchTest();
                Thread.Sleep(500);
            }
        }
    }
//</example>
```

As before, each counter contains its own display components: two text fields and a label that initially indicates that the counts are equivalent. These components are added to the panel of the **Sharing1** object in the **Sharing1** constructor.

Because a **TwoCounter** thread is started via a button press by the user, it's possible that **Start( )** could be called more than once. It's illegal for **Thread.Start( )** to be called more than once for a thread (an exception is thrown). You can see the machinery to prevent this in the **started** flag and the **Start( )** method. The **accessCountBox** in **Sharing1** keeps track of how many total accesses have been made on all **TwoCounter** threads. One way to do this would have been to have a static property that each **TwoCounter** could have incremented during **synchTest()**. Instead, we declared an **IncrementAccess()** delegate within **TwoCounter** that **Sharing1** provides as a parameter to the **TwoCounter** constructor.

In **run( )**, **count1** and **count2** are incremented and displayed in a manner that would seem to keep them identical. Then **Sleep( )** is called; without this call the UI becomes unresponsive because all the CPU time is being consumed within the loops.

The **synchTest( )** calls its **IncrementAccess** delegate and then performs the apparently superfluous activity of checking to see if **count1** is equivalent to **count2**; if they are not equivalent it sets the label to "Unsynched" to indicate this.

The **Watcher** class is a thread whose job is to call **synchTest( )** for all of the **TwoCounter** objects that are active. It does this by stepping through the array of **TwoCounters** passed to it by the **Sharing1** object. You can think of the **Watcher** as constantly peeking over the shoulders of the **TwoCounter** objects.

Sharing1 contains an array of **TwoCounter** objects that it initializes in its constructor and starts as threads when you press the "Start Threads" button. Later, when you press the "Begin Watching" button, one or more watchers are created and free to spy upon the unsuspecting **TwoCounter** threads.

By changing the **numCounters** and **numWatchers** values, which you can do at the command-line, you'll change the behavior of the program.

Here's the surprising part. In **TwoCounter.run( )**, the infinite loop is just repeatedly passing over the adjacent lines:

```
t1.Text = ++count1.ToString();
t2.Text = ++count2.ToString();
```

(as well as sleeping, but that's not important here). When you run the program, however, you'll discover that **count1** and **count2** will be observed (by the **Watcher**s) to be unequal at times! This is because of the nature of threads—they can be suspended at any time. So at times, the suspension occurs *between* the time count1 and count2 are incremented, and the **Watcher** thread happens to come along and perform the comparison at just this moment, thus finding the two counters to be different.

This example shows a fundamental problem with using threads. You never know when a thread might be run. Imagine sitting at a table with a fork, about to spear the last piece of food on your plate and as your fork reaches for it, the food suddenly vanishes (because your thread was suspended and another thread came in and stole the food). That's the problem that you're dealing with. Any time you rely on the state of an object being consistent, and that state can be manipulated by a different thread, you are vulnerable to this type of problem. This is what is known as a *race condition* (because your program's proper functioning is dependent on its thread winning the "race" to the resource). This type of bug (and all bugs relating to threading) is difficult to track down, as they will often slip under the radar of your unit testing code and appear and disappear depending on load, hardware and operating system differences, and the whimsy of the fates.

Preventing this kind of collision is simply a matter of putting a lock on a resource when one thread is relying on that resource. The first thread that accesses a resource locks it, and then the other threads cannot access that resource until it is unlocked, at which time another thread locks and uses it, etc. If the front seat of the car is the limited resource, the child who

shouts “Dibs!” asserts the lock. Using the Monitor class to prevent collisions @todo – confirm mechanism of Monitor and add Mutex sample code and Interlocked

Real-world programs have to share many types of resources – network sockets, database connections, sound channels, etc. By far the most common collisions, though, occur when some threads are changing the states of objects and other threads are relying on the state being consistent. This is the case with our **TwoCounter** and **Watcher** objects, where a thread controlled by **TwoCounter** increments the **count1** and **count2** variables, while a thread controlled by **Watcher** checks these variables for consistency.

The Monitor class in the namespace System.Threading helps prevent collisions over object state. Every **object** in C# has an associated “synchronization block” object which maintains a lock for that object and a queue of threads waiting to access the lock. The **Monitor** class is the public interface to the behind-the-scenes sync block implementation.

The method **Monitor.Enter(object o)** acts as gatekeeper – when a thread executes this line, the synchronization block for **o** is checked; if no one currently has the lock, the thread gets the lock and processing continues, but if another thread has already acquired the lock, the thread waits, or “blocks,” until the lock becomes available. When the critical section of code has been executed, the thread should call **Monitor.Exit(object o)** to release the lock. The next blocking thread will then be given a chance to obtain the lock.

Because you virtually always want to release the lock on a thread at the end of a critical section, even when throwing an **Exception**, calls to **Monitor.Enter()** and **Monitor.Exit()** are usually wrapped in a try block:

```
try{
    Monitor.Enter(o);
```

```
    //critical section
}finally{
    Monitor.Exit(o);
}
```

Synchronizing the counters

Armed with this technique it appears that the solution is at hand: we'll use the Monitor class to synchronize access to the counters. The following example is the same as the previous one, with the addition of **Monitor.Enter/Exit** calls at the two critical sections:

```
//<example>
///<chapter>c14</chapter>
///<program>Sharing2.cs</program>
// Using Monitor.enter() and exit()
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Threading;

public class Sharing1 : Form {
    private TextBox aCount = new TextBox();
    private Button start = new Button();
    private Button watch = new Button();

    private int accessCount = 0;
    public void incrementAccess() {
        accessCount++;
        aCount.Text = accessCount.ToString();
    }

    private int numCounters = 12;
    private int numWatchers = 15;

    private TwoCounter[] s;
    private Watcher[] w;

    public Sharing1() {
        ClientSize = new Size(450, 480);
        Panel p = new Panel();
        p.Size = new Size(400, 50);
```

```
        start.Click += new EventHandler(StartAllThreads);
        watch.Click += new
EventHandler(StartAllWatchers);

        aCount.Text = "0";
        aCount.Location = new Point(10, 10);
        start.Text = "Start threads";
        start.Location = new Point(110, 10);
        watch.Text = "Begin watching";
        watch.Location = new Point(210, 10);

        p.Controls.Add(start);
        p.Controls.Add(watch);
        p.Controls.Add(aCount);

        s = new TwoCounter[numCounters];
        for (int i = 0; i < s.Length; i++) {
            s[i] = new TwoCounter(new
TwoCounter.IncrementAccess(incrementAccess));
            s[i].Location = new Point(10, 50 +
s[i].Height * i);
            Controls.Add(s[i]);
        }

        this.Closed += new EventHandler(StopAllThreads);

        Controls.Add(p);
    }

    public void StartAllThreads(Object sender, EventArgs
args) {
        for (int i = 0; i < s.Length; i++)
            s[i].Start();
    }

    public void StopAllThreads(Object sender, EventArgs
args){
        for(int i = 0; i < s.Length; i++){
            if(s[i] != null){
                s[i].Stop();
```

```
            }
        }
        for(int i = 0; i < w.Length; i++){
            if(w[i] != null){
                w[i].Stop();
            }
        }
    }

    public void StartAllWatchers(Object sender, EventArgs
args) {
        w = new Watcher[numWatchers];
        for (int i = 0; i < numWatchers; i++)
            w[i] = new Watcher(s);
    }

    public static void Main(string[] args) {
        Sharing1 app = new Sharing1();
        if (args.Length > 0) {
            app.numCounters = SByte.Parse(args[0]);
            if (args.Length == 2) {
                app.numWatchers = SByte.Parse(args[1]);
            }
        }
        Application.Run(app);
    }
}

    class TwoCounter : Panel {
        private bool started = false;
        private Label t1;
        private Label t2;
        private Label lbl;
        private Thread t;

        private int count1 = 0, count2 = 0;
        public delegate void IncrementAccess();
        IncrementAccess del;

        // Add the display components
        public TwoCounter(IncrementAccess del) {
```

```
            this.del = del;

            this.Size = new Size(350, 30);
            this.BorderStyle = BorderStyle.Fixed3D;
            t1 = new Label();
            t1.Location = new Point(10, 10);
            t2 = new Label();
            t2.Location = new Point(110, 10);
            lbl = new Label();
            lbl.Text = "Count1 == Count2";
            lbl.Location = new Point(210, 10);
            Controls.AddRange(new Control[]{t1, t2,
lbl});

            //Initialize the Thread
            t = new Thread(new ThreadStart(run));
        }
        public void Start() {
            if (!started) {
                started = true;
                t.Start();
            }
        }

        public void Stop(){
            t.Abort();
        }
        public void run() {
            while (true) {
                try{
                    Monitor.Enter(this);
                    t1.Text = (++count1).ToString();
                    t2.Text = (++count2).ToString();
                }finally{
                    Monitor.Exit(this);
                }
                Thread.Sleep(500);
            }
        }
        public void synchTest() {
            del();
```

```
                try
                {
                      Monitor.Enter(this);
                      if (count1 != count2)
                      {
                            lbl.Text = "Unsynched";
                      }
                }
                finally
                {
                      Monitor.Exit(this);
                }
          }
     }

     class Watcher {
          TwoCounter[] s;
          Thread t;

          public Watcher(TwoCounter[] s) {
               this.s = s;
               t = new Thread(new ThreadStart(run));
               t.Start();
          }
          public void run() {
               while (true) {
                    for (int i = 0; i < s.Length; i++)
                        s[i].synchTest();
                    Thread.Sleep(500);
               }
          }
          public void Stop(){
               t.Abort();
          }
     }
//</example>
```

You'll notice that *both* **run( )** and **synchTest( )** call **Monitor.Enter()** and **Exit().** If you use the **Monitor** only on **run()**, the **Watcher** threads

calling **synchTest()** will happily read the state of the **TwoCounter** even while the **TwoCounter** thread has entered the critical section and has placed the object in an inconsistent state by incrementing **Counter1** but not yet changing **Counter2**.

There's nothing magic about thread synchronization – it's the manipulation of the **TwoCounter** instance's sync block via the **Monitor** that does the work. All synchronization depends on programmer diligence: every piece of code that can place an object in an inconsistent state or that relies on an object being in a consistent state must be wrapped in an appropriate block.

"lock" blocks – a shortcut for using the Monitor

In order to save you some time typing, C# provides the **lock** keyword, which creates a guarded codeblock exactly equivalent to the **try{ Monitor.Enter()...}finally{Monitor.Exit()}** idiom. Instead of all that typing, you use the **lock(object)** keyword and specify a code block to be protected:

```
lock(this){
     //critical section
}
```

Although **lock()**'s form makes it appear to be a call to a base class method (implemented by **object** presumably), it's really just syntactic sugar. The choice of **lock** as a keyword may be a little misleading, in that you may expect that a "locked" object would be automatically threadsafe. This is not so; **lock** says that the *current* thread will act appropriately. If all threads follow the rules, things will work out for the best, but if another piece of code has a reference to the "locked" object, it may mistakenly manipulate the object without ever using **Monitor**. It's like taking a number for service at a bakery – a fine idea that breaks down as soon as someone misbehaves through laziness or ignorance.

Choosing what to monitor

Monitor.Enter() and **Exit()** can be passed any object for synchronization. It is best to use **this** – an inability to use **this** for

synchronization (perhaps because you have mutually exclusive types of inconsistency to guard against) is a "code smell" that indicates that your object may be trying to do too many things at once and may be best broken up into smaller classes. For instance, in the TwoCounter class, we could place **counter1** and **counter2** in an inner **class**, add thread-safe accessors and methods, and achieve our goal without ever locking the entire **TwoPanel** instance:

```
//<example>
///<chapter>c14</chapter>
///<program>Sharing3.cs</program>
// Refactoring for finer granularity
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Threading;

public class Sharing3 : Form {
    private TextBox aCount = new TextBox();
    private Button start = new Button();
    private Button watch = new Button();

    private int accessCount = 0;
    public void incrementAccess() {
        accessCount++;
        aCount.Text = accessCount.ToString();
    }

    private int numCounters = 12;
    private int numWatchers = 15;

    private TwoCounter[] s;
    private Watcher[] w;

    public Sharing1() {
        ClientSize = new Size(450, 480);
        Panel p = new Panel();
        p.Size = new Size(400, 50);

        start.Click += new EventHandler(StartAllThreads);
```

```
        watch.Click += new
EventHandler(StartAllWatchers);

        aCount.Text = "0";
        aCount.Location = new Point(10, 10);
        start.Text = "Start threads";
        start.Location = new Point(110, 10);
        watch.Text = "Begin watching";
        watch.Location = new Point(210, 10);

        p.Controls.Add(start);
        p.Controls.Add(watch);
        p.Controls.Add(aCount);

        s = new TwoCounter[numCounters];
        for (int i = 0; i < s.Length; i++) {
            s[i] = new TwoCounter(new
TwoCounter.IncrementAccess(incrementAccess));
            s[i].Location = new Point(10, 50 +
s[i].Height * i);
            Controls.Add(s[i]);
        }

        this.Closed += new EventHandler(StopAllThreads);

        Controls.Add(p);
    }

    public void StartAllThreads(Object sender, EventArgs
args) {
        for (int i = 0; i < s.Length; i++)
            s[i].Start();
    }

    public void StopAllThreads(Object sender, EventArgs
args){
        for(int i = 0; i < s.Length; i++){
            if(s[i] != null){
                s[i].Stop();
            }
        }
```

```
        for(int i = 0; i < w.Length; i++){
            if(w[i] != null){
                w[i].Stop();
            }
        }
    }

    public void StartAllWatchers(Object sender, EventArgs
args) {
        w = new Watcher[numWatchers];
        for (int i = 0; i < numWatchers; i++)
            w[i] = new Watcher(s);
    }

    public static void Main(string[] args) {
        Sharing3 app = new Sharing3();
        if (args.Length > 0) {
            app.numCounters = SByte.Parse(args[0]);
            if (args.Length == 2) {
                app.numWatchers = SByte.Parse(args[1]);
            }
        }
        Application.Run(app);
    }
}

class TwoCounter : Panel
{
     private bool started = false;
     private Label t1;
     private Label t2;
     private Label lbl;
     private Thread t;

     class Counter
     {
          private int c1 = 0;
          private int c2 = 0;
          public void Increment()
          {
               lock(this)
```

```
                {
                        ++c1;
                        ++c2;
                }
            }
            public int Count1
            {
                get{ lock(this){ return c1;}
                }
            }
            public int Count2
            {
                get{ lock(this){ return c2; }
                }
            }
        }
        private Counter counter = new Counter();
          public delegate void IncrementAccess();
          IncrementAccess del;

          // Add the display components
          public TwoCounter(IncrementAccess del) {
              this.del = del;

              this.Size = new Size(350, 30);
              this.BorderStyle = BorderStyle.Fixed3D;
              t1 = new Label();
              t1.Location = new Point(10, 10);
              t2 = new Label();
              t2.Location = new Point(110, 10);
              lbl = new Label();
              lbl.Text = "Count1 == Count2";
              lbl.Location = new Point(210, 10);
              Controls.AddRange(new Control[]{t1, t2,
lbl});

              //Initialize the Thread
              t = new Thread(new ThreadStart(run));
          }
          public void Start() {
              if (!started) {
```

```
                started = true;
                t.Start();
            }
        }

        public void Stop(){
            t.Abort();
        }
        public void run() {
            while (true) {
                        counter.Increment();
                        t1.Text =
counter.Count1.ToString();
                        t2.Text =
counter.Count2.ToString();
                Thread.Sleep(500);
            }
        }
        public void synchTest() {
            del();
                 if(counter.Count1 != counter.Count2)
                        {
                              lbl.Text = "Unsynched";
                        }
        }
    }

    class Watcher {
        TwoCounter[] s;
        Thread t;

        public Watcher(TwoCounter[] s) {
            this.s = s;
            t = new Thread(new ThreadStart(run));
            t.Start();
        }
        public void run() {
            while (true) {
                for (int i = 0; i < s.Length; i++)
                    s[i].synchTest();
                Thread.Sleep(500);
```

```
            }
        }
        public void Stop(){
            t.Abort();
        }
    }
//</example>
```

The **Counter** class, an inner class of **TwoPanel** is now "thread-safe," by removing any chance that an external object can place it in an inconsistent state. Incrementing the counter integers is done inside the **Increment** method, with a **lock**ed critical section, and access to the integers is done via the **Count1** and **Count2** properties, which also synchronize against the Monitor to ensure that they cannot be read until the **Increment** critical section has exited (and, undesirably, also ensure that **Count1** and **Count2** cannot be read simultaneously by two different threads – a small penalty typical of the design decisions made when developing multithreaded apps).

Sometimes, locking **this** doesn't seem like the right idea. When an object contains some resource, and especially when that resource is a container of some sort, such as a **Collection** or a **Stream** or an **Image**, it is common to want to perform some operation across some subset of that resource without worrying about whether some other thread will change the resource halfway through your operation. In a situation like this, it's common to lock the resource, not **this.** I'm of two minds on this: on the one hand, the principle of coupling leads to the thought that if the only thing that's vulnerable to being placed in an inconsistent state is the resource, then lock the resource, as locking **this** unnecessarily couples **this** and the resource. On the other hand, the principle of cohesion leads us to think that if one portion of the methods and resources in an instance are vulnerable to race conditions, but other methods and resources in the instance aren't, then maybe we ought to refactor. This is what we did with our **TwoPanel** class, splitting the initial class into two, and I think **Sharing3** is clearly a superior design to **Sharing2.**

Where to monitor

An object's thread-safety is entirely a function of the object's state. A class without any static or instance variables, consisting solely of methods, is inherently threadsafe. Any variables or resources that affect whether your object is in a consistent state should be **private** or **protected** and only available to outside objects by way of properties or methods. It's just foolish to ever allow direct references to these critical resources from external objects. If instead you create properties and methods which consistently use the **Monitor** class (or the equivalent **lock** blocks), you'll save yourself considerable headaches when it comes to locating and debugging threading problems.

Threads and Collections

The Collection classes in .NET's **System.Collections** namespace are not threadsafe and behavior is "undefined" when collisions occur. This program illustrates the issue:

```
//<example>
using System;
using System.Collections;
using System.Threading;

class SyncCol1{
    public static void Main(){
        int iThreads = 25;
        SyncCol1 sc1 = new SyncCol1(iThreads);
    }

    int iThreads;
    SortedList myList;
    public SyncCol1(int iThreads){
        this.iThreads = iThreads;
        myList = new SortedList();
        TimedWrite(myList);
        myList = SortedList.Synchronized(new
SortedList());
        TimedWrite(myList);
    }
```

```
    public void TimedWrite(SortedList myList){
        WriterThread.ExceptionCount = 0;
        WriterThread[] writerThreads = new
WriterThread[iThreads];
        DateTime start = DateTime.Now;
        for(int i = 0; i < iThreads; i++){
            writerThreads[i] = new WriterThread(myList,
i);
            writerThreads[i].Start();
        }
        WaitForAllThreads(writerThreads);
        DateTime stop = DateTime.Now;
        TimeSpan elapsed = stop - start;
        System.Console.WriteLine("Synchronized List: " +
myList.IsSynchronized);
        System.Console.WriteLine(iThreads + " * 5000 = "
+ myList.Count + "? " + (myList.Count == (iThreads *
5000)));
        System.Console.WriteLine("Number of exceptions
thrown: " + WriterThread.ExceptionCount);
        System.Console.WriteLine("Time of calculation = "
+ elapsed);
    }

    public void WaitForAllThreads(WriterThread[] ts){
        for(int i = 0; i < ts.Length; i++){
            while(ts[i].Finished == false){
                Thread.Sleep(1000);
            }
        }
    }
}

class WriterThread {
    static int iExceptionsThrown = 0;
    public static int ExceptionCount{
        get{ return iExceptionsThrown; }
        set{ iExceptionsThrown = value; }
    }

    Thread t;
```

```
    SortedList theList;

    public WriterThread(SortedList theList, int i){
        t = new Thread(new ThreadStart(WriteThread));
        t.IsBackground = true;
        t.Name = "Writer[" + i.ToString() + "]";
        this.theList = theList;
    }
    public bool Finished{
        get{ return isFinished; }
    }
    bool isFinished = false;

    public void WriteThread(){
        for(int loop = 0; loop < 5000; loop++){
            String elName = t.Name + loop.ToString();
            try{
                theList.Add(elName, elName);
            }catch(Exception ){
               ++iExceptionsThrown;
            }
        }
        isFinished = true;
        t.Abort();
    }
    public void Start(){
        t.Start();
    }
}
//</example>
```

The **Main()** creates a **SyncCol1** class with a parameter indicating how many threads to simultaneously write to a collection. A **SortedList** is created and passed to the **TimedWrite** method. This method sets the static variable **ExceptionCount** of the **WriterThread** class to 0 and creates an array of **WriterThread**s. The **WriterThread** constructor takes the list and a variable. Each **WriterThread** creates a new thread, whose processing is delegated to the **WriterThread.WriteThread()** method. The **Background** property of the **WriterThread**'s thread is set to true. A program won't exit until all it's non-background threads have ended. Being able to create background "daemon" threads is very

convenient, especially in a GUI, where the user can request a program closure at any time.

After the **WriterThread** constructor returns, the next line of **TimedWrite()** calls the **Start()** method, which in turn starts the inner thread, which in turn delegates processing to **WriteThread(). WriteThread()** loops 5,000 times, each time creating a new name (such as "Writer[12]237") and attempting to add that to **theList**. A **SortedArray** is backed by two stores – one to store the values and another to store a sorted list of keys (the keys may or may not be the same as the values). The sorted list is stored in a data structure called a "balanced tree" (more specifically, I believe it is backed by a "red-black" tree). "Balanced trees" have the property that looking up any item happens in a time proportional to the logarithm of the total number of items stored.

When speaking of algorithmic efficiency, it is best to ignore the actual number of instructions involved and only concentrate on the order of the exponential increase associated with scaling the algorithm. The actual number of steps, after all, varies from programming language to programming language and computer speeds increase constantly. So what *really* matters boils down to what's called "Big O" notation – a red-black tree does a lookup in O(log n) steps. An unsorted array, on the other hand, requires O(n) steps to search (on average, you'll find an item in n/2 comparisons, but for the purposes of "Big O" conversations, you drop everything but the exponent). But if you want really fast retrieval, the **Hashtable** class does a lookup in O(1) steps – what's called "constant time." Of course, as Table *x* shows, for every advantage, there's a disadvantage.

Collection Class	Insert	Delete	Lookup
ArrayList	O(1)		
BitArray			
Hashtable			O(1)
Queue			

SortedList	O(log n)	O(log n)	O(log n)
Stack			

The large majority of effort that programmers expend on efficiency and optimization and performance ignores Big O efficiency and concentrates on trivial aspects of removing an intermediate assignment here, packing twice as much data in a variable over there, and so forth. Don't fall for this mistake – if it's not Big O efficiency, it's almost certainly not worth paying attention to.

On what I promise is the last tangent before getting back to threads, the big magic in quantum computers is that they reduce the algorithmic cost of rotating an *n*-dimensional hypersphere from $O(n^2)$ to O(1). If my analysis is correct, this means that any problem you can solve by manipulating a Rubik's cube while wearing a blindfold can be solved by a two-qubit quantum computer in constant time. Have at it, gang.

Back to **WriterThread.WriteThread()**. The call to **Add()** an element to the list is wrapped in a catch block. Since we are ignoring the details of the exception and only recording how many exceptions were thrown, the **catch** statement does not specify a variable name for the caught exception. Once the loop is finished, we set the **Finished** property of the **WriterThread,** kill the Thread, and return. Back in the **SyncCol1** class, the main application thread goes through the array, checking to see if it's finished. If it's not, the main thread goes to sleep for 1,000 seconds before checking again. When all the **WriterThread**'s are **Finished**, the **WriteThread()** writes some data on the experiment and returns.

After the initial call with a regular **SortedList**, we create a new **SortedList** and pass it to the static method **SortedList.Synchronized()**. All the Collections have this static method, which creates a new, thread-safe Collection. To be clear, the program creates a total of 3 **SortedList**s: the one for the initial run through **WriteThread()**, a second anonymous one, which is used as the parameter to **SortedList.Synchronize()**, which returns a third one. After Chapter **#stream chapter #**, you should recognize the Decorator pattern in play.

When you run this program, you'll see that the first run, with a plain **SortedList** throws a large number of exceptions (if you have a sufficiently speedy computer, you may get no exceptions, but if you increase the number of threads, eventually you'll run into trouble), while the list produced by **Synchronized()** adds all the data flawlessly. You'll also see why Collections aren't synchronized by default: the thread-safe list takes something like 5-8 times the duration to complete. (If you said "But it's the same Big O!" give yourself a gold star. If you said, "But ignorant hacks would confuse library performance with language performance and compare thread-safe collections to non-thread-safe collections, and on the basis of simplistic benchmarks write that C# has a performance problem, just as they did with Java!" give yourself a platinum star.) You can find out if a Collection is synchronized or not by examining it's **Synchronized** property, as **TimedWrite()** does during its status-writing lines.

If, instead of using a list produced by **SortedList.Synchronized()**, you put a **lock(theList)** block around the **Add()** call, you'll get exception-less behavior on both runs as well. Curiously, if you do this, the synchronized list seems to always outperform the unsynchronized list by a small margin!

In general, though, the challenge of working with collections and threads is not the thread-safety of the underlying collection, but the inherent challenge of objects being added, deleted, or changed by threads while your current thread tries to deal with the collection as a single logical unit. For instance, in an object with an instance object called **myCollection**, whether the Collection is **Synchronized** or not, the lines

```
int i = myCollection.Count;
Object o = myCollection[i];
```

are inherently thread-unsafe because another thread might have removed element **i** before the second line is executed. If the class obeys the recommendation that the only references to a critical resource like **myCollection** are internal, any method that accesses **myCollection** can simply **lock(this)** and achieve thread-safety.

It's not always possible to design classes that don't expose internal instance or static collections, but give it a hard try before giving up on the

attempt. Can you **Clone** the collection? Use the Proxy pattern to return, not the collection itself, but an interface to your own threadsafe methods? If not, be prepared for some long debugging sessions, because it's a good bet that any time you open the door to multithreading defects, someone will introduce them.

Threads, Delegates, and Events

Now that you understand the Monitor class, lock blocks, and issues with collection classes, you can take another look at Delegates and Events. For any class that generates Events (and remember, Events aren't just for GUIs anymore), you must assume that it will run in a multithreaded environment. Similarly, when writing EventHandlers or Delegates, you must assume that the source of the Event may *already* have changed by the time that the delegate is called. Consider this program, which attempts to trap the position of a fast moving mouse:

@start here

15: XML

Schemas and DataSets

16: Web Services

A: C# For Java Programmers

B: C# For Visual Basic Programmers

C: C# Programming Guidelines

This appendix contains suggestions to help guide you in performing low-level program design, and in writing code.

Naturally, these are guidelines and not rules. The idea is to use them as inspirations, and to remember that there are occasional situations where you need to bend or break a rule.

Design

1. **Elegance always pays off**. In the short term it might seem like it takes much longer to come up with a truly graceful solution to a problem, but when it works the first time and easily adapts to new situations instead of requiring hours, days, or months of struggle, you'll see the rewards (even if no one can measure them). Not only does it give you a program that's easier to build and debug, but it's also easier to understand and maintain, and that's where the financial value lies. This point can take some experience to understand, because it can appear that you're not being productive while you're making a piece of code elegant. Resist the urge to hurry; it will only slow you down.

17. **First make it work, then make it fast**. This is true even if you are certain that a piece of code is really important and that it will be a principal bottleneck in your system. Don't do it. Get the system going first with as simple a design as possible. Then if it isn't going fast enough, profile it. You'll almost always discover that "your" bottleneck isn't the problem. Save your time for the really important stuff.

18. **Remember the "divide and conquer" principle**. If the problem you're looking at is too confusing, try to imagine what the basic operation of the program would be, given the existence of a magic "piece" that handles the hard parts. That "piece" is an object—write the code that uses the object, then look at the object and encapsulate *its* hard parts into other objects, etc.

19. **Separate the class creator from the class user (*client programmer*)**. The class user is the "customer" and doesn't need or want to know what's going on behind the scenes of the class. The class creator must be the expert in class design and write the class so that it can be used by the most novice programmer possible, yet still work robustly in the application. Library use will be easy only if it's transparent.

20. **When you create a class, attempt to make your names so clear that comments are unnecessary**. Your goal should be to make the client programmer's interface conceptually simple. To this end, use method overloading when appropriate to create an intuitive, easy-to-use interface.

21. **Your analysis and design must produce, at minimum, the classes in your system, their public interfaces, and their relationships to other classes, especially base classes**. If your design methodology produces more than that, ask yourself if all the pieces produced by that methodology have value over the lifetime of the program. If they do not, maintaining them will cost you. Members of development teams tend not to maintain anything that does not contribute to their productivity; this is a fact of life that many design methods don't account for.

22. **Automate everything**. Write the test code first (before you write the class), and keep it with the class. Automate the running of your tests through a makefile or similar tool. This way, any changes can be automatically verified by running the test code, and you'll immediately discover errors. Because you know that you have the safety net of your test framework, you will be bolder about making sweeping changes when you discover the need. Remember that the greatest improvements in languages come from the built-in testing provided by

type checking, exception handling, etc., but those features take you only so far. You must go the rest of the way in creating a robust system by filling in the tests that verify features that are specific to your class or program.

23. **Write the test code first (before you write the class) in order to verify that your class design is complete**. If you can't write test code, you don't know what your class looks like. In addition, the act of writing the test code will often flush out additional features or constraints that you need in the class—these features or constraints don't always appear during analysis and design. Tests also provide example code showing how your class can be used.

24. **All software design problems can be simplified by introducing an extra level of conceptual indirection**. This fundamental rule of software engineering[1] is the basis of abstraction, the primary feature of object-oriented programming.

25. **An indirection should have a meaning** (in concert with guideline 9). This meaning can be something as simple as "putting commonly used code in a single method." If you add levels of indirection (abstraction, encapsulation, etc.) that don't have meaning, it can be as bad as not having adequate indirection.

26. **Make classes as atomic as possible**. Give each class a single, clear purpose. If your classes or your system design grows too complicated, break complex classes into simpler ones. The most obvious indicator of this is sheer size: if a class is big, chances are it's doing too much and should be broken up.
    Clues to suggest redesign of a class are:
    1) A complicated switch statement: consider using polymorphism.
    2) A large number of methods that cover broadly different types of operations: consider using several classes.
    3) A large number of member variables that concern broadly different characteristics: consider using several classes.

[1] Explained to me by Andrew Koenig.

27. **Watch for long argument lists**. Method calls then become difficult to write, read, and maintain. Instead, try to move the method to a class where it is (more) appropriate, and/or pass objects in as arguments.

28. **Don't repeat yourself**. If a piece of code is recurring in many methods in derived classes, put that code into a single method in the base class and call it from the derived-class methods. Not only do you save code space, you provide for easy propagation of changes. Sometimes the discovery of this common code will add valuable functionality to your interface.

29. **Watch for *switch* statements or chained *if-else* clauses**. This is typically an indicator of *type-check coding*, which means you are choosing what code to execute based on some kind of type information (the exact type may not be obvious at first). You can usually replace this kind of code with inheritance and polymorphism; a polymorphic method call will perform the type checking for you, and allow for more reliable and easier extensibility.

30. **From a design standpoint, look for and separate things that change from things that stay the same**. That is, search for the elements in a system that you might want to change without forcing a redesign, then encapsulate those elements in classes. You can learn significantly more about this concept in *Thinking in Patterns with Java*, downloadable at *www.BruceEckel.com*.

31. **Don't extend fundamental functionality by subclassing**. If an interface element is essential to a class it should be in the base class, not added during derivation. If you're adding methods by inheriting, perhaps you should rethink the design.

32. **Less is more**. Start with a minimal interface to a class, as small and simple as you need to solve the problem at hand, but don't try to anticipate all the ways that your class *might* be used. As the class is used, you'll discover ways you must expand the interface. However, once a class is in use you cannot shrink the interface without disturbing client code. If you need to add more methods, that's fine; it won't disturb code, other than forcing recompiles. But even if new methods replace the functionality of old ones, leave the existing

interface alone (you can combine the functionality in the underlying implementation if you want). If you need to expand the interface of an existing method by adding more arguments, create an overloaded method with the new arguments; this way you won't disturb any existing calls to the existing method.

33. **Read your classes aloud to make sure they're logical**. Refer to the relationship between a base class and derived class as "is-a" and member objects as "has-a."

34. **When deciding between inheritance and composition, ask if you need to upcast to the base type**. If not, prefer composition (member objects) to inheritance. This can eliminate the perceived need for multiple base types. If you inherit, users will think they are supposed to upcast.

35. **Use data members for variation in value and method overriding for variation in behavior**. That is, if you find a class that uses state variables along with methods that switch behavior based on those variables, you should probably redesign it to express the differences in behavior within subclasses and overridden methods.

36. **Watch for overloading**. A method should not conditionally execute code based on the value of an argument. In this case, you should create two or more overloaded methods instead.

37. **Use exception hierarchies**—preferably derived from specific appropriate classes in the standard Java exception hierarchy. The person catching the exceptions can then catch the specific types of exceptions, followed by the base type. If you add new derived exceptions, existing client code will still catch the exception through the base type.

38. **Sometimes simple aggregation does the job**. A "passenger comfort system" on an airline consists of disconnected elements: seat, air conditioning, video, etc., and yet you need to create many of these in a plane. Do you make private members and build a whole new interface? No—in this case, the components are also part of the public interface, so you should create public member objects. Those objects

have their own private implementations, which are still safe. Be aware that simple aggregation is not a solution to be used often, but it does happen.

39. **Consider the perspective of the client programmer and the person maintaining the code**. Design your class to be as obvious as possible to use. Anticipate the kind of changes that will be made, and design your class so that those changes will be easy.

40. **Watch out for "giant object syndrome."** This is often an affliction of procedural programmers who are new to OOP and who end up writing a procedural program and sticking it inside one or two giant objects. With the exception of application frameworks, objects represent concepts in your application, not the application.

41. **If you must do something ugly, at least localize the ugliness inside a class**.

42. **If you must do something nonportable, make an abstraction for that service and localize it within a class**. This extra level of indirection prevents the nonportability from being distributed throughout your program. (This idiom is embodied in the *Bridge* Pattern).

43. **objects should not simply hold some data**. They should also have well-defined behaviors. (Occasionally, "data objects" are appropriate, but only when used expressly to package and transport a group of items when a generalized container is innappropriate.)

44. **Choose composition first when creating new classes from existing classes**. You should only used inheritance if it is required by your design. If you use inheritance where composition will work, your designs will become needlessly complicated.

45. **Use inheritance and method overriding to express differences in behavior, and fields to express variations in state**. An extreme example of what not to do is inheriting different classes to represent colors instead of using a "color" field.

46. **Watch out for *variance***. Two semantically different objects may have identical actions, or responsibilities, and there is a natural

temptation to try to make one a subclass of the other just to benefit from inheritance. This is called variance, but there's no real justification to force a superclass/subclass relationship where it doesn't exist. A better solution is to create a general base class that produces an interface for both as derived classes—it requires a bit more space, but you still benefit from inheritance, and will probably make an important discovery about the design.

47. **Watch out for *limitation* during inheritance**. The clearest designs add new capabilities to inherited ones. A suspicious design removes old capabilities during inheritance without adding new ones. But rules are made to be broken, and if you are working from an old class library, it may be more efficient to restrict an existing class in its subclass than it would be to restructure the hierarchy so your new class fits in where it should, above the old class.

48. **Use design patterns to eliminate "naked functionality."** That is, if only one object of your class should be created, don't bolt ahead to the application and write a comment "Make only one of these." Wrap it in a singleton. If you have a lot of messy code in your main program that creates your objects, look for a creational pattern like a factory method in which you can encapsulate that creation. Eliminating "naked functionality" will not only make your code much easier to understand and maintain, it will also make it more bulletproof against the well-intentioned maintainers that come after you.

49. **Watch out for "analysis paralysis."** Remember that you must usually move forward in a project before you know everything, and that often the best and fastest way to learn about some of your unknown factors is to go to the next step rather than trying to figure it out in your head. You can't know the solution until you *have* the solution. Java has built-in firewalls; let them work for you. Your mistakes in a class or set of classes won't destroy the integrity of the whole system.

50. **When you think you've got a good analysis, design, or implementation, do a walkthrough**. Bring someone in from outside your group—this doesn't have to be a consultant, but can be

someone from another group within your company. Reviewing your work with a fresh pair of eyes can reveal problems at a stage when it's much easier to fix them, and more than pays for the time and money "lost" to the walkthrough process.

Implementation

51. **In general, follow the Microsoft coding conventions**. These are available at
#ref# (the code in this book follows these conventions as much as I was able). These are used for what constitutes arguably the largest body of code that the largest number of C# programmers will be exposed to. If you doggedly stick to the coding style you've always used, you will make it harder for your reader. Whatever coding conventions you decide on, ensure they are consistent throughout the project.

52. **Whatever coding style you use, it really does make a difference if your team (and even better, your company) standardizes on it**. This means to the point that everyone considers it fair game to fix someone else's coding style if it doesn't conform. The value of standardization is that it takes less brain cycles to parse the code, so that you can focus more on what the code means.

53. **Follow standard capitalization rules**. Capitalize the first letter of class names and non-private methods and properties. The first letter of private fields, methods, and objects (references) should be lowercase. All identifiers should run their words together, and capitalize the first letter of all intermediate words. For example:
ThisIsAClassName
thisIsAPrivateMethodOrFieldName
Capitalize *all* the letters of **static final** primitive identifiers that have constant initializers in their definitions. This indicates they are compile-time constants.

54. **Don't create your own "decorated" private data member names**. This is usually seen in the form of prepended underscores and characters. Hungarian notation is the worst example of this, where you attach extra characters that indicate data type, use,

location, etc., as if you were writing assembly language and the compiler provided no extra assistance at all. These notations are confusing, difficult to read, and unpleasant to enforce and maintain. Let classes and packages do the name scoping for you.

55. **Follow a "canonical form"** when creating a class for general-purpose use. Include definitions for **equals( )**, **hashCode( )**, **toString( )**, **clone( )** (implement **Cloneable**), and implement **Comparable** and **Serializable**.

56. **Use the JavaBeans "get," "set," and "is" naming conventions** for methods that read and change **private** fields, even if you don't think you're making a JavaBean at the time. Not only does it make it easy to use your class as a Bean, but it's a standard way to name these kinds of methods and so will be more easily understood by the reader.

57. **For each class you create, consider including a *static public test( )* that contains code to test that class**. You don't need to remove the test code to use the class in a project, and if you make any changes you can easily rerun the tests. This code also provides examples of how to use your class.

58. **Sometimes you need to inherit in order to access *protected* members of the base class**. This can lead to a perceived need for multiple base types. If you don't need to upcast, first derive a new class to perform the protected access. Then make that new class a member object inside any class that needs to use it, rather than inheriting.

59. **If two classes are associated with each other in some functional way (such as containers and iterators), try to make one an inner class of the other**. This not only emphasizes the association between the classes, but it allows the class name to be reused within a single package by nesting it within another class.

60. **Anytime you notice classes that appear to have high coupling with each other, consider the coding and maintenance improvements you might get by using inner**

classes. The use of inner classes will not uncouple the classes, but rather make the coupling explicit and more convenient.

61. **Don't fall prey to premature optimization**. This way lies madness. In particular, don't worry about writing (or avoiding) native methods, making some methods **final**, or tweaking code to be efficient when you are first constructing the system. Your primary goal should be to prove the design, unless the design requires a certain efficiency.
62. **Keep scopes as small as possible so the visibility and lifetime of your objects are as small as possible**. This reduces the chance of using an object in the wrong context and hiding a difficult-to-find bug. For example, suppose you have a container and a piece of code that iterates through it. If you copy that code to use with a new container, you may accidentally end up using the size of the old container as the upper bound of the new one. If, however, the old container is out of scope, the error will be caught at compile-time.
63. **Use the containers in the .NET Framework SDK**. Become proficient with their use and you'll greatly increase your productivity.
64. **For a program to be robust, each component must be robust**. Use all the tools provided by C#: access control, exceptions, type checking, and so on, in each class you create. That way you can safely move to the next level of abstraction when building your system.
65. **Prefer compile-time errors to run-time errors**. Try to handle an error as close to the point of its occurrence as possible. Prefer dealing with the error at that point to throwing an exception. Catch any exceptions in the nearest handler that has enough information to deal with them. Do what you can with the exception at the current level; if that doesn't solve the problem, rethrow the exception.
66. **Watch for long method definitions**. Methods should be brief, functional units that describe and implement a discrete part of a class interface. A method that is long and complicated is difficult and expensive to maintain, and is probably trying to do too much all by

itself. If you see such a method, it indicates that, at the least, it should be broken up into multiple methods. It may also suggest the creation of a new class. Small methods will also foster reuse within your class. (Sometimes methods must be large, but they should still do just one thing.)

67. **Keep things as "*private* as possible."** Once you publicize an aspect of your library (a method, a class, a field), you can never take it out. If you do, you'll wreck somebody's existing code, forcing them to rewrite and redesign. If you publicize only what you must, you can change everything else with impunity, and since designs tend to evolve this is an important freedom. In this way, implementation changes will have minimal impact on derived classes. Privacy is especially important when dealing with multithreading—only **private** fields can be protected against un-**synchronized** use.

68. **Use comments liberally, and use the comment-documentation syntax to produce your program documentation**. However, the comments should add geniune meaning to the code; comments that only reiterate what the code is clearly expressing are annoying. Note that the typical verbose detail of Java class and method names reduce the need for as many comments.

69. **Avoid using "magic numbers"**—which are numbers hard-wired into code. These are a nightmare if you need to change them, since you never know if "100" means "the array size" or "something else entirely." Instead, create a constant with a descriptive name and use the constant identifier throughout your program. This makes the program easier to understand and much easier to maintain.

70. **When creating constructors, consider exceptions**. In the best case, the constructor won't do anything that throws an exception. In the next-best scenario, the class will be composed and inherited from robust classes only, so they will need no cleanup if an exception is thrown. Otherwise, you must clean up composed classes inside a **finally** clause. If a constructor must fail, the appropriate action is to throw an exception, so the caller doesn't continue blindly, thinking that the object was created correctly.

71. **If your class requires any cleanup when the client programmer is finished with the object, make your class implement IDisposable** tk.

72. **When you are creating a fixed-size container of objects, transfer them to an array**—especially if you're returning this container from a method. This way you get the benefit of the array's compile-time type checking, and the recipient of the array might not need to cast the objects in the array in order to use them.

73. **Choose *interfaces* over *abstract* classes**. If you know something is going to be a base class, your first choice should be to make it an **interface**, and only if you're forced to have method definitions or member variables should you change it to an **abstract** class. An **interface** talks about what the client wants to do, while a class tends to focus on (or allow) implementation details.

74. **Inside constructors, do only what is necessary to set the object into the proper state**. Actively avoid calling other methods (except for **final** methods) since those methods can be overridden by someone else to produce unexpected results during construction. (See Chapter 7 for details.) Smaller, simpler constructors are less likely to throw exceptions or cause problems.

75. **Watch out for accidental overloading**. If you attempt to override a base-class method and you don't quite get the spelling right, you'll end up adding a new method rather than overriding an existing method. However, this is perfectly legal, so you won't get any error message from the compiler or run-time system—your code simply won't work correctly.

76. **Watch out for premature optimization**. First make it work, then make it fast—but only if you must, and only if it's proven that there is a performance bottleneck in a particular section of your code. Unless you have used a profiler to discover a bottleneck, you will probably be wasting your time. The hidden cost of performance tweaks is that your code becomes less understandable and maintainable.

77. **Remember that code is read much more than it is written**. Clean designs make for easy-to-understand programs, but comments,

detailed explanations, and examples are invaluable. They will help both you and everyone who comes after you. If nothing else, the frustration of trying to ferret out useful information from the online Java documentation should convince you.

D: Resources

Software

tk

Books

C#

tk

Analysis & design

tk

Management & Process

tk

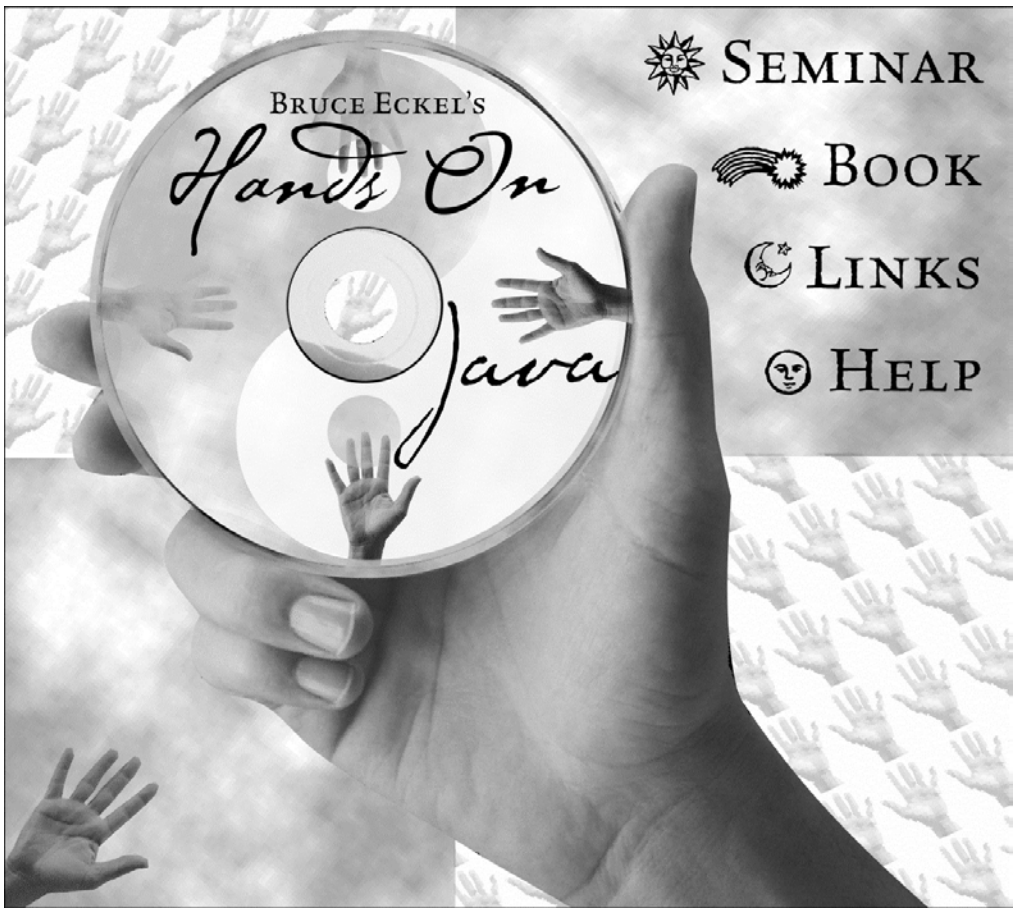
Index

Please note that some names will be duplicated in capitalized form. Following C# style, the capitalized names refer to C# classes, while lowercase names refer to a general concept.**Error! No index entries found.**

Bruce Eckel's
HANDS-ON
JAVA™
SEMINAR
Learn the programming language of the World Wide Web
In this step-by-step introduction each carefully-chosen subject is covered in a lecture followed by hands-on programming exercises.
This course is for you if you can follow basic code examples written in C language syntax.
WWW.BRUCEECKEL.COM

Check www.BruceEckel.com
for in-depth details
and the date and location
of the next
Hands-On Java Seminar

- Based on this book
- Taught by Bruce Eckel
- Personal attention from Bruce Eckel and his seminar assistants
- Includes in-class programming exercises
- Intermediate/Advanced seminars also offered
- Hundreds have already enjoyed this seminar—see the Web site for their testimonials



Bruce Eckel's Hands-On Java Seminar
Multimedia CD
It's like coming to the seminar!
Available at *www.BruceEckel.com*

- The *Hands-On Java Seminar* captured on a Multimedia CD!
- Overhead slides and synchronized audio voice narration for all the lectures. Just play it to see and hear the lectures!
- Created and narrated by Bruce Eckel.
- Based on the material in this book.
- Demo lecture available at *www.BruceEckel.com*

End-User License Agreement for Microsoft Software

IMPORTANT-READ CAREFULLY: This Microsoft End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and Microsoft Corporation for the Microsoft software product included in this package, which includes computer software and may include associated media, printed materials, and "online" or electronic documentation ("SOFTWARE PRODUCT"). The SOFTWARE PRODUCT also includes any updates and supplements to the original SOFTWARE PRODUCT provided to you by Microsoft. By installing, copying, downloading, accessing or otherwise using the SOFTWARE PRODUCT, you agree to be bound by the terms of this EULA. If you do not agree to the terms of this EULA, do not install, copy, or otherwise use the SOFTWARE PRODUCT.

SOFTWARE PRODUCT LICENSE

The SOFTWARE PRODUCT is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. The SOFTWARE PRODUCT is licensed, not sold.
1. GRANT OF LICENSE. This EULA grants you the following rights:
1.1 License Grant. Microsoft grants to you as an individual, a personal nonexclusive license to make and use copies of the SOFTWARE PRODUCT for the sole purposes of evaluating and learning how to use the SOFTWARE PRODUCT, as may be instructed in accompanying publications or documentation. You may install the software on an unlimited number of computers provided that you are the only individual using the SOFTWARE PRODUCT.
1.2 Academic Use. You must be a "Qualified Educational User" to use the SOFTWARE PRODUCT in the manner described in this section. To determine whether you are a Qualified Educational User, please contact the Microsoft Sales Information Center/One Microsoft Way/Redmond, WA 98052-6399 or the Microsoft subsidiary serving your country. If you are a Qualified Educational User, you may either:
(i) exercise the rights granted in Section 1.1, OR
(ii) if you intend to use the SOFTWARE PRODUCT solely for instructional purposes in connection with a class or other educational program, this EULA grants you the following alternative license models:
(A) Per Computer Model. For every valid license you have acquired for the SOFTWARE PRODUCT, you may install a single copy of the SOFTWARE PRODUCT on a single computer for access and use by an unlimited number of

student end users at your educational institution, provided that all such end users comply with all other terms of this EULA, OR
(B) Per License Model. If you have multiple licenses for the SOFTWARE PRODUCT, then at any time you may have as many copies of the SOFTWARE PRODUCT in use as you have licenses, provided that such use is limited to student or faculty end users at your educational institution and provided that all such end users comply with all other terms of this EULA. For purposes of this subsection, the SOFTWARE PRODUCT is "in use" on a computer when it is loaded into the temporary memory (i.e., RAM) or installed into the permanent memory (e.g., hard disk, CD ROM, or other storage device) of that computer, except that a copy installed on a network server for the sole purpose of distribution to other computers is not "in use". If the anticipated number of users of the SOFTWARE PRODUCT will exceed the number of applicable licenses, then you must have a reasonable mechanism or process in place to ensure that the number of persons using the SOFTWARE PRODUCT concurrently does not exceed the number of licenses.
2. DESCRIPTION OF OTHER RIGHTS AND LIMITATIONS.
• Limitations on Reverse Engineering, Decompilation, and Disassembly. You may not reverse engineer, decompile, or disassemble the SOFTWARE PRODUCT, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation.
• Separation of Components. The SOFTWARE PRODUCT is licensed as a single product. Its component parts may not be separated for use on more than one computer.
• Rental. You may not rent, lease or lend the SOFTWARE PRODUCT.
• Trademarks. This EULA does not grant you any rights in connection with any trademarks or service marks of Microsoft.
• Software Transfer. The initial user of the SOFTWARE PRODUCT may make a one-time permanent transfer of this EULA and SOFTWARE PRODUCT only directly to an end user. This transfer must include all of the SOFTWARE PRODUCT (including all component parts, the media and printed materials, any upgrades, this EULA, and, if applicable, the Certificate of Authenticity). Such transfer may not be by way of consignment or any other indirect transfer. The transferee of such one-time transfer must agree to comply with the terms of this EULA, including the obligation not to further transfer this EULA and SOFTWARE PRODUCT.
• No Support. Microsoft shall have no obligation to provide any product support for the SOFTWARE PRODUCT.
• Termination. Without prejudice to any other rights, Microsoft may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In

such event, you must destroy all copies of the SOFTWARE PRODUCT and all of its component parts.
3. COPYRIGHT. All title and intellectual property rights in and to the SOFTWARE PRODUCT (including but not limited to any images, photographs, animations, video, audio, music, text, and "applets" incorporated into the SOFTWARE PRODUCT), the accompanying printed materials, and any copies of the SOFTWARE PRODUCT are owned by Microsoft or its suppliers. All title and intellectual property rights in and to the content which may be accessed through use of the SOFTWARE PRODUCT is the property of the respective content owner and may be protected by applicable copyright or other intellectual property laws and treaties. This EULA grants you no rights to use such content. All rights not expressly granted are reserved by Microsoft.
4. BACKUP COPY. After installation of one copy of the SOFTWARE PRODUCT pursuant to this EULA, you may keep the original media on which the SOFTWARE PRODUCT was provided by Microsoft solely for backup or archival purposes. If the original media is required to use the SOFTWARE PRODUCT on the COMPUTER, you may make one copy of the SOFTWARE PRODUCT solely for backup or archival purposes. Except as expressly provided in this EULA, you may not otherwise make copies of the SOFTWARE PRODUCT or the printed materials accompanying the SOFTWARE PRODUCT.
5. U.S. GOVERNMENT RESTRICTED RIGHTS. The SOFTWARE PRODUCT and documentation are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable. Manufacturer is Microsoft Corporation/One Microsoft Way/Redmond, WA 98052-6399.
6. EXPORT RESTRICTIONS. You agree that you will not export or re-export the SOFTWARE PRODUCT, any part thereof, or any process or service that is the direct product of the SOFTWARE PRODUCT (the foregoing collectively referred to as the "Restricted Components"), to any country, person, entity or end user subject to U.S. export restrictions. You specifically agree not to export or re-export any of the Restricted Components (i) to any country to which the U.S. has embargoed or restricted the export of goods or services, which currently include, but are not necessarily limited to Cuba, Iran, Iraq, Libya, North Korea, Sudan and Syria, or to any national of any such country, wherever located, who intends to transmit or transport the Restricted Components back to such country; (ii) to any end-user who you know or have reason to know will utilize the Restricted Components in the design, development or production of nuclear, chemical or biological weapons; or (iii) to any end-user who has been

prohibited from participating in U.S. export transactions by any federal agency of the U.S. government. You warrant and represent that neither the BXA nor any other U.S. federal agency has suspended, revoked, or denied your export privileges.
7. NOTE ON JAVA SUPPORT. THE SOFTWARE PRODUCT MAY CONTAIN SUPPORT FOR PROGRAMS WRITTEN IN JAVA. JAVA TECHNOLOGY IS NOT FAULT TOLERANT AND IS NOT DESIGNED, MANUFACTURED, OR INTENDED FOR USE OR RESALE AS ON-LINE CONTROL EQUIPMENT IN HAZARDOUS ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS IN THE OPERATION OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION OR COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL, DIRECT LIFE SUPPORT MACHINES, OR WEAPONS SYSTEMS, IN WHICH THE FAILURE OF JAVA TECHNOLOGY COULD LEAD DIRECTLY TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE.

MISCELLANEOUS

If you acquired this product in the United States, this EULA is governed by the laws of the State of Washington.
If you acquired this product in Canada, this EULA is governed by the laws of the Province of Ontario, Canada. Each of the parties hereto irrevocably attorns to the jurisdiction of the courts of the Province of Ontario and further agrees to commence any litigation which may arise hereunder in the courts located in the Judicial District of York, Province of Ontario.
If this product was acquired outside the United States, then local law may apply.
Should you have any questions concerning this EULA, or if you desire to contact Microsoft for any reason, please contact
Microsoft, or write: Microsoft Sales Information Center/One Microsoft Way/Redmond, WA 98052-6399.

LIMITED WARRANTY

LIMITED WARRANTY. Microsoft warrants that (a) the SOFTWARE PRODUCT will perform substantially in accordance with the accompanying written materials for a period of ninety (90) days from the date of receipt, and (b) any Support Services provided by Microsoft shall be substantially as described in applicable written materials provided to you by Microsoft, and Microsoft support engineers will make commercially reasonable efforts to solve any problem. To the extent allowed by applicable law, implied warranties on the SOFTWARE PRODUCT, if any, are limited to ninety (90) days. Some states/jurisdictions do

not allow limitations on duration of an implied warranty, so the above limitation may not apply to you.

CUSTOMER REMEDIES. Microsoft's and its suppliers' entire liability and your exclusive remedy shall be, at Microsoft's option, either (a) return of the price paid, if any, or (b) repair or replacement of the SOFTWARE PRODUCT that does not meet Microsoft's Limited Warranty and that is returned to Microsoft with a copy of your receipt. This Limited Warranty is void if failure of the SOFTWARE PRODUCT has resulted from accident, abuse, or misapplication. Any replacement SOFTWARE PRODUCT will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer. Outside the United States, neither these remedies nor any product support services offered by Microsoft are available without proof of purchase from an authorized international source.

NO OTHER WARRANTIES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, MICROSOFT AND ITS SUPPLIERS DISCLAIM ALL OTHER WARRANTIES AND CONDITIONS, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, WITH REGARD TO THE SOFTWARE PRODUCT, AND THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES. THIS LIMITED WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHERS, WHICH VARY FROM STATE/JURISDICTION TO STATE/JURISDICTION.

LIMITATION OF LIABILITY. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL MICROSOFT OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE PRODUCT OR THE FAILURE TO PROVIDE SUPPORT SERVICES, EVEN IF MICROSOFT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY CASE, MICROSOFT'S ENTIRE LIABILITY UNDER ANY PROVISION OF THIS EULA SHALL BE LIMITED TO THE GREATER OF THE AMOUNT ACTUALLY PAID BY YOU FOR THE SOFTWARE PRODUCT OR U.S.$5.00; PROVIDED, HOWEVER, IF YOU HAVE ENTERED INTO A MICROSOFT SUPPORT SERVICES AGREEMENT, MICROSOFT'S ENTIRE LIABILITY REGARDING SUPPORT SERVICES SHALL BE GOVERNED BY THE TERMS OF THAT AGREEMENT. BECAUSE SOME STATES/JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

0495 Part No. 64358

LICENSE AGREEMENT FOR MindView, Inc.'s
Thinking in C: Foundations for Java & C++ CD ROM
by Chuck Allison
This CD is provided together with the book "Thinking in Java, 2nd edition."

READ THIS AGREEMENT BEFORE USING THIS "Thinking in C: Foundations for C++ & Java" (Hereafter called "CD"). BY USING THE CD YOU AGREE TO BE BOUND BY THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS AND CONDITIONS OF THIS AGREEMENT, IMMEDIATELY RETURN THE UNUSED CD FOR A FULL REFUND OF MONIES PAID, IF ANY.

©2000 MindView, Inc. All rights reserved. Printed in the U.S.

SOFTWARE REQUIREMENTS
The purpose of this CD is to provide the Content, not the associated software necessary to view the Content. The Content of this CD is in HTML for viewing with Microsoft Internet Explorer 4 or newer, and uses Microsoft Sound Codecs available in Microsoft's Windows Media Player for Windows. It is your responsibility to correctly install the appropriate Microsoft software for your system.

The text, images, and other media included on this CD ("Content") and their compilation are licensed to you subject to the terms and conditions of this Agreement by MindView, Inc., having a place of business at 5343 Valle Vista, La Mesa, CA 91941. Your rights to use other programs and materials included on the CD are also governed by separate agreements distributed with those programs and materials on the CD (the "Other Agreements"). In the event of any inconsistency between this Agreement and the Other Agreements, this Agreement shall govern. By using this CD, you agree to be bound by the terms and conditions of this Agreement. MindView, Inc. owns title to the Content and to all intellectual property rights therein, except insofar as it contains materials that are proprietary to third-party suppliers. All rights in the Content except those expressly granted to you in this Agreement are reserved to MindView, Inc. and such suppliers as their respective interests may appear.

1. LIMITED LICENSE
MindView, Inc. grants you a limited, nonexclusive, nontransferable license to use the Content on a single dedicated computer (excluding network servers). This Agreement and your rights hereunder shall automatically terminate if you fail to comply with any provisions of this Agreement or any of the Other

Agreements. Upon such termination, you agree to destroy the CD and all copies of the CD, whether lawful or not, that are in your possession or under your control.

2. ADDITIONAL RESTRICTIONS

a. You shall not (and shall not permit other persons or entities to) directly or indirectly, by electronic or other means, reproduce (except for archival purposes as permitted by law), publish, distribute, rent, lease, sell, sublicense, assign, or otherwise transfer the Content or any part thereof.

b. You shall not (and shall not permit other persons or entities to) use the Content or any part thereof for any commercial purpose or merge, modify, create derivative works of, or translate the Content.

c. You shall not (and shall not permit other persons or entities to) obscure MindView's or its suppliers copyright, trademark, or other proprietary notices or legends from any portion of the Content or any related materials.

3. PERMISSIONS

a. Except as noted in the Contents of the CD, you must treat this software just like a book. However, you may copy it onto a computer to be used and you may make archival copies of the software for the sole purpose of backing up the software and protecting your investment from loss. By saying, "just like a book," MindView, Inc. means, for example, that this software may be used by any number of people and may be freely moved from one computer location to another, so long as there is no possibility of its being used at one location or on one computer while it is being used at another. Just as a book cannot be read by two different people in two different places at the same time, neither can the software be used by two different people in two different places at the same time.

b. You may show or demonstrate the un-modified Content in a live presentation, live seminar, or live performance as long as you attribute all material of the Content to MindView, Inc.

c. Other permissions and grants of rights for use of the CD must be obtained directly from MindView, Inc. at http://www.MindView.net. (Bulk copies of the CD may also be purchased at this site.)

DISCLAIMER OF WARRANTY

The Content and CD are provided "AS IS" without warranty of any kind, either express or implied, including, without limitation, any warranty of

merchantability and fitness for a particular purpose. The entire risk as to the results and performance of the CD and Content is assumed by you. MindView, Inc. and its suppliers assume no responsibility for defects in the CD, the accuracy of the Content, or omissions in the CD or the Content. MindView, Inc. and its suppliers do not warrant, guarantee, or make any representations regarding the use, or the results of the use, of the product in terms of correctness, accuracy, reliability, currentness, or otherwise, or that the Content will meet your needs, or that operation of the CD will be uninterrupted or error-free, or that any defects in the CD or Content will be corrected. MindView, Inc. and its suppliers shall not be liable for any loss, damages, or costs arising from the use of the CD or the interpretation of the Content. Some states do not allow exclusion or limitation of implied warranties or limitation of liability for incidental or consequential damages, so all of the above limitations or exclusions may not apply to you.

In no event shall MindView, Inc. or its suppliers' total liability to you for all damages, losses, and causes of action (whether in contract, tort, or otherwise) exceed the amount paid by you for the CD.
MindView, Inc., and Prentice-Hall, Inc. specifically disclaim the implied warrantees of merchantability and fitness for a particular purpose. No oral or written information or advice given by MindView, Inc., Prentice-Hall, Inc., their dealers, distributors, agents or employees shall create a warrantee. You may have other rights, which vary from state to state.
Neither MindView, Inc., Bruce Eckel, Chuck Allison, Prentice-Hall, nor anyone else who has been involved in the creation, production or delivery of the product shall be liable for any direct, indirect, consequential, or incidental damages (including damages for loss of business profits, business interruption, loss of business information, and the like) arising out of the use of or inability to use the product even if MindView, Inc., has been advised of the possibility of such damages. Because some states do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to you.
This CD is provided as a supplement to the book "Thinking in Java 2nd edition." The sole responsibility of Prentice-Hall will be to provide a replacement CD in the event that the one that came with the book is defective. This replacement warrantee shall be in effect for a period of sixty days from the purchase date. MindView, Inc. does not bear any additional responsibility for the CD.
NO TECHNICAL SUPPORT IS PROVIDED WITH THIS CD ROM
The following are trademarks of their respective companies in the U.S. and may be protected as trademarks in other countries: Sun and the Sun Logo,

Sun Microsystems, Java, all Java-based names and logos and the Java Coffee Cup are trademarks of Sun Microsystems; Internet Explorer, the Windows Media Player, DOS, Windows 95, and Windows NT are trademarks of Microsoft.

Thinking in C: Foundations for Java & C++

Multimedia Seminar-on-CD ROM

©2000 MindView, Inc. All rights reserved.

WARNING: BEFORE OPENING THE DISC PACKAGE, CAREFULLY READ THE TERMS AND CONDITIONS OF THE LICENSE AGREEMENT & WARANTEE LIMITATION ON THE PREVIOUS PAGES.

The CD ROM packaged with this book is a multimedia seminar consisting of synchronized slides and audio lectures. The goal of this seminar is to introduce you to the aspects of C that are necessary for you to move on to C++ or Java, leaving out the unpleasant parts that C programmers must deal with on a day-to-day basis but that the C++ and Java languages steer you away from. The CD also contains this book in HTML form along with the source code for the book.

This CD ROM will work with Windows (with a sound system). However, you must:

1. Install the most recent version of Microsoft's Internet Explorer. Because of the features provided on the CD, it will NOT work with Netscape Navigator. **The Internet Explorer software for Windows 9X/NT is included on the CD.**
2. Install Microsoft's *Windows Media Player*. **The Media Player software for Windows 9X/NT is included on the CD.** You can also go to **http://www.microsoft.com/windows/mediaplayer** and follow the instructions or links there to download and install the Media Player for your particular platform.
3. At this point you should be able to play the lectures on the CD. Using the Internet Explorer Web browser, open the file **Install.html** that you'll find on the CD. This will introduce you to the CD and provide further instructions about the use of the CD.